Foundations of Dependent Interoperability

Pierre-Evariste Dagand, Nicolas Tabareau, Éric Tanter

Under consideration for publication in J. Functional Programming

Foundations of Dependent Interoperability

PIERRE-ÉVARISTE DAGAND
Sorbonne Universités—UPMC Univ Paris 06
CNRS—Inria—LIP6 UMR 7606

NICOLAS TABAREAU
Inria

ÉRIC TANTER
PLEIAD lab—Computer Science Dept (DCC)
University of Chile
(*e-mail:* {pierre-evariste.dagand,nicolas.tabareau}@inria.fr, etanter@dcc.uchile.cl)

Abstract

Full-spectrum dependent types promise to enable the development of correct-by-construction software. However, even certified software needs to interact with simply-typed or untyped programs, be it to perform system calls, or to use legacy libraries. Trading static guarantees for runtime checks, the *dependent interoperability* framework provides a mechanism by which simply-typed values can safely be coerced to dependent types and, conversely, dependently-typed programs can defensively be exported to a simply-typed application.

In this article, we give a semantic account of dependent interoperability. Our presentation relies on and is guided by a pervading notion of type equivalence, whose importance has been emphasized in recent work on homotopy type theory. Specifically, we develop the notions of *type-theoretic partial Galois connections* as a key foundation for dependent interoperability, which accounts for the partiality of the coercions between types.

We explore the applicability of both type-theoretic Galois connections and anticonnections in the setting of dependent interoperability. A partial Galois connection enforces a translation of dependent types to runtime checks that are both sound and complete with respect to the invariants encoded by dependent types. Conversely, picking an anticonnection instead lets us induce weaker, sound conditions that can amount to more efficient runtime checks.

Our framework is developed in Coq; it is thus constructive and verified in the strictest sense of the terms. Using our library, users can specify domain-specific partial connections between data structures. Our library then takes care of the (sometimes, heavy) lifting that leads to interoperable programs. It thus becomes possible, as we shall illustrate, to internalize and hand-tune the extraction of dependently-typed programs to interoperable OCaml programs within Coq itself.

This work was partially funded by the CoqHoTT ERC Grant 637339, by FONDECYT Project 1150017 and the Émergence(s) program of Paris.

1 Introduction

Dependent interoperability is a pragmatic approach to building reliable software systems, where the adoption of dependent types may be incremental or limited to certain components. The safe interaction between the simple and dependent type disciplines relies on a *marshaling* mechanism to convert values from one world to the other, as well as *dynamic checks*, to ensure that the properties stated by the dependent type system are respected by the simply-typed values injected in dependent types.[1]

Following Osera *et al.* (2012), we illustrate the typical use cases of dependent interoperability using the stereotypical example of simply-typed lists and dependently-typed vectors. For conciseness, we fix the element type of lists and vectors to natural numbers and use the type synonyms `List` $\mathbb{N}$ and `Vec` $\mathbb{N}$ `n`, where `n` denotes the length of the vector.

Using a simply-typed library in a dependently-typed context. One may want to reuse an existing simply-typed library from a dependently-typed context. For instance, the list library may provide a function `sum`: `List` $\mathbb{N} \to \mathbb{N}$ that computes the sum of the elements of a list (with 0 the neutral element). To reuse this existing function on vectors requires lifting the `sum` function to the type $\forall$ `n`. `Vec` $\mathbb{N}$ `n` $\to \mathbb{N}$. Note that this scenario only requires losing information about the vector used as argument, so no dynamic check is needed, only a marshaling to reconstruct the corresponding list value. If the simply-typed function returns a list, *e.g.* `rev`: `List` $\mathbb{N} \to$ `List` $\mathbb{N}$, then the target dependent type might entail a dynamic check on the returned value.

Using a dependently-typed library in a simply-typed context. Dually, one may want to apply a function that operates on vectors to plain lists. For instance a sorting function of type $\forall$ `n`. `Vec` $\mathbb{N}$ `n` $\to$ `Vec` $\mathbb{N}$ `n` could be reused at type `List` $\mathbb{N} \to$ `List` $\mathbb{N}$. Note that this case requires synthesizing the index `n`. Also, because the simply-typed argument flows to the dependently-typed world, a dynamic check might be needed. Indeed, the function `Vector.tl`: $\forall$ `n`. `Vec` $\mathbb{N}$ (`n+1`) $\to$ `Vec` $\mathbb{N}$ `n`, should trigger a runtime error if it is called on an empty list. On the return value, however, no error can be raised.

Verifying simply-typed components. One can additionally use dependent interoperability to dynamically verify properties of simply-typed components through a dependently-typed interface and then going back to their simply-typed interface, thereby combining both scenarios above. For instance, we can specify that a function `List.tl`: `List` $\mathbb{N} \to$ `List` $\mathbb{N}$ should behave as a function of type $\forall$ `n`. `Vec` $\mathbb{N}$ (`n+1`) $\to$ `Vec` $\mathbb{N}$ `n` by first lifting it to this rich type, and then recasting it back to a simply-typed function `tl`' of type `List` $\mathbb{N}$ $\to$ `List` $\mathbb{N}$. While both `tl` and `tl`' have the same type and "internal" definition, `tl`' will raise an error if called with an empty list; additionally, if the argument list is not empty, `tl`' will dynamically check that it returns a list that is one element smaller than its input. This is similar to dependent contracts in untyped languages (Findler & Felleisen, 2002).

[1] In this article, we use the term "simply typed" to mean "non-dependently typed", *i.e.* we do not rule out parametric polymorphism.

Program extraction. Several dependently-typed programming languages use program extraction as a means to obtain (fast(er)) executables. Coq is the most prominent example, but more recent languages like Agda, Idris, and F* also integrate extraction mechanisms, at different degrees (*e.g.* extraction in F* is the only mechanism to actually run programs, while in Agda it is mostly experimental at this point).

Dependent interoperability is crucial for extraction, if extracted components are meant to openly interact with other components written in the target language. While Tanter and Tabareau (2015) address the question of protecting the extracted components from inputs that violate conditions expressed as subset types in Coq[2], the situation can be even worse with type dependencies, because extracting dependent structures typically introduces unsafe operations; hence invalid inputs can easily produce segmentation faults. By directly enforcing structural invariants through indexing, dependent datatypes have become a key tool for functional programmers (McKinna, 2006), be they Coq, Agda or even Haskell and OCaml programmers. Understanding and offering an interoperable extraction for such types is thus essential.

Consider the following example adapted from the book *Certified Programming with Dependent Types* (Chlipala, 2013), in which the types of the instructions for a stack machine are explicit about their effect on the size of the stack:

```
Inductive dinstr: ℕ → ℕ → Type :=
| IConst: ∀ n, ℕ → dinstr n (S n)
| IPlus: ∀ n, dinstr (S (S n)) (S n).
```

An `IConst` instruction operates on any stack of size `n`, and produces a stack of size `(S n)`, where `S` is the successor constructor of ℕ. Similarly, an `IPlus` instruction consumes two values from the stack (hence the stack size must have the form `(S (S n))`), and pushes back one value. A dependently-typed stack of depth `n` is represented by nested pairs:

```
Fixpoint dstack (n: ℕ): Type :=
 match n with
  | O ⇒ unit
  | S n' ⇒ ℕ * dstack n'
 end.
```

The `exec` function, which executes an instruction on a given stack and returns the new stack can be defined as follows:

```
Definition exec n m (i: dinstr n m): dstack n → dstack m :=
 match i with
  | IConst n ⇒ fun s ⇒ (n, s)
  | IPlus ⇒ fun s ⇒
        let '(arg1, (arg2, s)) := s in (arg1 + arg2, s)
 end.
```

[2] In Coq terminology, a subset type is a type refined by a proposition—this is also known in the literature as refinement type (Xi & Pfenning, 1998; Rondon *et al.*, 2008; Knowles & Flanagan, 2010).

Of special interest is the fact that in the `IPlus` case, the stack `s` is deconstructed by directly grabbing the top two elements through pattern matching, without having to check that the stack has at least two elements— this is guaranteed by the type dependencies.

Because such type dependencies are absent in OCaml, the `exec` function is extracted into a function that ignores its stack size arguments, and relies on unsafe coercions:

```
(* exec: int -> int -> dinstr -> dstack -> dstack *)
let exec _ _ i s =
  match i with
  | IConst (n, _) -> Obj.magic (n, s)
  | IPlus _ ->
    let (arg1, s1) = Obj.magic s in
    let (arg2, s2) = s1 in Obj.magic ((add arg1 arg2), s2)
```

The `dstack` indexed type from Coq cannot be expressed in mini-ML (Letouzey, 2004) (which amounts to the subset of OCaml to which Coq programs are extracted), so the extracted code defines the (plain) type `dstack` as:

```
type dstack = Obj.t
```

where `Obj.t` is the abstract internal representation type of any value. Therefore, the type system has in fact no information at all about stacks: the unsafe coercion `Obj.magic` (of type $\forall a \forall b. a \to b$) is used to convert from and to this internal representation type. The dangerous coercion is the one in the `IPlus` case, when coercing `s` to a nested pair of depth at least 2. Consequently, applying `exec` with an improper stack yields a segmentation fault:

```
# exec 0 0 (IPlus 0) [1;2];;
- : int list = [3]
# exec 0 0 (IPlus 0) [];;
Segmentation fault: 11
```

Dependent interoperability helps in such scenarios by making it possible to lift dependent structures—and functions that operate on them—to types that are faithfully expressible in the type system of the target language in a safe way, *i.e.* embedding dynamic checks that protects extracted code from executing unsafe operations under violated assumptions.[3] We come back to this stack machine example and how to protect the extracted `exec` function in Section 6.

Contributions. In this article, we present a verified dependent interoperability layer for Coq that exploits the notion of type equivalence from Homotopy Type Theory (HoTT). In particular, our contributions are the following:

- Using type equivalences as a guiding principle, we give a unified treatment of *type-theoretic (partial) Galois connections* between programs (Section 3). Doing so, we build a conceptual as well as practical framework for relating indexed and simple types;

[3] Note that some unsafe executions can be produced by using impure functions as arguments to functions extracted from Coq—because referential transparency is broken. Designing an adequate protection mechanism to address such scenarios is a separate, interesting research challenge.

- By carefully segregating the computational and logical content of indexed types, we introduce a notion of *dependent connection* (Section 4) that identifies first-order transformations between indexed and simple datatypes. In particular, we show that an indexed type can be seen as the combination of its underlying computational representation and a runtime check that its associated logical invariant holds;
- To deal with programs, we extend the presentation to a higher-order setting (Section 5). Using the type class mechanism of Coq, we provide a generic library for establishing partial connections of dependently-typed programs;
- We illustrate our methodology through a concrete application: extracting an interoperable, certified interpreter (Section 6). Aside from exercising our library, this example is also a performance in homotopic theorem proving (Section 7);
- Finally, we introduce a notion of *anticonnection* (Section 8) to allow programmers to implement more efficient, though incomplete, run-time checks. We also study the relationship between partial connections and partial anticonnections, which governs the modular design of our library.

This article is thus deeply entrenched at the crossroad between mathematics and programming. From the former, we borrow and introduce some homotopic definitions as well as proof patterns. For the latter, we are led to design interoperable—yet safe—programs and are willing to trade static guarantees against runtime checks. We therefore chose to rely on Coq notations and mechanisms to account for the mathematical as well as programming aspects of this work: doing so enables us to formalize as well as program our constructs in the mathematical and programming language that is Coq. Section 3 provides many self-contained examples for a reader to get acquainted with the subset of Coq manipulated in the rest of the paper.

Coq Formalization. The full Coq formalization, with documentation, is available at

`http://coqhott.github.io/DICoq/`.

It has been developed using the 8.6 release of Coq (The Coq Development Team, 2016). It consists of about 4000 lines of code, of which 500 lines correspond to two applications: a safe `tail` operator for lists and an interpreter for a type-safe stack machine. It relies on two postulates: an axiomatization of truncated types (Awodey & Bauer, 2004) (whose innocuousness is discussed in Section 3.3) and the functional extensionality axiom (whose carefully controlled use is detailed in Section 5.1).

Relation to prior publication. This article extends and refines an article that appeared in the proceedings of ICFP 2016 (Dagand *et al.*, 2016). Compared to this previous work, we have made the following improvements:

- We have clarified the overall conceptual framework by identifying the notion of "partial type equivalence" (Dagand *et al.*, 2016) as a type-theoretic form of Galois connection. This has guided some notational improvements (such as replacing the misleadingly symmetric notation $\cong_K$ with the asymmetric symbol $\lesssim_K$) and reorganizing our definitions around the concept of Galois connection (Section 3 and Section 4).

- Having identified the structuring role of Galois connections in our earlier work, we have been led to adopt a monotone presentation by default, which provides tighter relations between programs (Section 3). This also allows us to provide a separate treatment of anticonnections, which we had implicitly adopted in our earlier work, to specific situations where the completeness of the runtime coercions can or must be sacrificed (Section 8). In turn, this leads to a rationalized presentation of *checkable* properties as the counterpart to *decidable* properties;
- We have made significant efforts to simplify our type-theoretic structures. In particular, our earlier definition of partial orders took place in the proof-relevant universe `Prop`, which led to spurious and unnatural coherence conditions on the proofs establishing ordering results. To address this mismatch, we have defined partial orders in `HProp`, stating that ordering proofs are irrelevant. This led to dramatic simplifications in the later definitions of Galois connections, removing the need for coherence conditions between sections and retractions. Another benefit is that we have been able to prove that being a type connection is itself a proof-irrelevant property: if there exists one, it is unique.
- Following our conceptual simplifications, we have made several terminological adjustments to distinguish the underlying mathematical structures from the transformative mechanisms. A type-theoretic (partial) Galois connection is thus called a "(partial) type connection" as it justifies a *structural* property offered by the existence of two functions relating both types. These functions, in turn, are called *coercions*, as they represent a *mechanism* for translating values across those types. We consistently maintain this distinction throughout the article.
- We have replaced the misleading term of "cast" of values along type connections by the more appropriate term "coercion": a mechanism such as ours is indeed a dynamic operation transforming the underlying representation of data, whilst the term "cast" wrongly suggests that the underlying structure could remain unchanged. In the process, we have done away with our earlier `Cast` monad, which had nothing specific to do with coercing computations, so we chose to rename it to `TError`, along the lines of the `Error` monad of Haskell.

2 Methodology

In this section we motivate our formalism, *type-theoretic (partial) Galois connections*, through the familiar relation between the dependently-typed `Vec` $\mathbb{N}$ `n` and its simply-typed counterpart `List` $\mathbb{N}$. Our objective is to define a pair of (partial) functions mediating between vectors and lists, allowing us to switch safely between both representations. We also want to precisely characterize the relationship between both functions.

We notice that there is a total embedding (Univalent Foundations Program, 2013)

$$\texttt{forget n}\colon \texttt{Vec}\ \mathbb{N}\ \texttt{n} \to \texttt{List}\ \mathbb{N}$$

from vectors of size n to lists. The challenge is thus to define a (necessarily partial) function mapping lists to vectors. To do so, we adopt a two-step process. First, we exploit the standard notion of type equivalence (Section 3.1) to relate the type `Vec` $\mathbb{N}$ `n` to its image by `forget`:

$$\texttt{im forget n} := \{\ \texttt{l}\colon \texttt{List}\ \mathbb{N}\ \&\ \exists\ \texttt{v}\colon \texttt{Vec}\ \mathbb{N}\ \texttt{n},\ \texttt{forget n v} = \texttt{l}\ \}$$

that explicitly segregates the computational content of vectors—"a list" —from its logical content—"whose length is equal to n". Indeed, the property $\exists$ v: Vec $\mathbb{N}$ n, forget n v = l is logically equivalent to the property length l = n.

Because forget is an embedding, the restriction of forget to its image establishes a type equivalence (denoted $\simeq$) between an inductive family and a subset type:

$$\texttt{Vec}\ \mathbb{N}\ \texttt{n} \simeq \texttt{im forget n} \qquad \text{for all } n \in \mathbb{N}$$

Second, we must relate the subset type im forget n with the simple type List $\mathbb{N}$. Once again, there is a total embedding from the dependent type to the simple one: it is in fact the first projection of the Σ-type! Conversely, going from simple types to subset types is a potentially partial operation. We model partiality using the standard monadic framework (Section 3.3) with a monadic composition $\circ_K$, an identity creturn, a lifting lift from pure to monadic operations and a flat partial order relation such that $\bot \preceq$ a and Some a $\preceq$ Some b *iff* a = b.

The back-translation from lists to im forget n is thus a partial function, denoted with the arrow $\rightharpoonup$:

$$\texttt{make n}\colon \texttt{List}\ \mathbb{N} \rightharpoonup \{\ \texttt{l}\colon \texttt{List}\ \mathbb{N}\ \&\ \exists\ \texttt{v}\colon \texttt{Vec}\ \mathbb{N}\ \texttt{n},\ \texttt{forget n v} = \texttt{l}\ \}$$

that may fail, *esp.* if the length of the input list is different from n:

$$\begin{aligned} \forall\ \texttt{n}, \forall\ \texttt{l}\colon \texttt{List}\ \mathbb{N}, \quad & ((\texttt{lift}\ \pi_1) \circ_K (\texttt{make n}))\ \texttt{l} = \texttt{Some l} \\ & \vee\ ((\texttt{lift}\ \pi_1) \circ_K (\texttt{make n}))\ \texttt{l} = \texttt{Fail} \end{aligned} \tag{1}$$

and must amount to an identity function when the input list is of the right length:

$$\forall\ \texttt{n}, \forall\ \texttt{l}\colon \texttt{im forget n}, \texttt{Some l} = \texttt{make n}\ (\pi_1\ \texttt{l}) \tag{2}$$

Translated in the monadic language, these two specifications become:

$$\begin{aligned} &\forall\ \texttt{n}, (\texttt{lift}\ \pi_1) \circ_K (\texttt{make n}) \preceq \texttt{creturn} && \text{by (1)} \\ &\forall\ \texttt{n}, \qquad \texttt{creturn} \preceq (\texttt{make n}) \circ_K (\texttt{lift}\ \pi_1) && \text{by (2)} \end{aligned}$$

whereby we recognize a type-theoretic and partial version of a monotone Galois connection (Section 3.5)—or *partial connection* for short, conventionally written A $\lesssim_K$ B. As hinted at by its name, this definition is based on an adjunction, whose origin can in fact be traced back to the notion of type equivalence in homotopy type theory.

Note that having creturn $\preceq$ f means that f is total and is, in fact, the identity function. Here, this means that one direction (from subset type to simple type) never fails: given an index n, and a list enriched with the property that it corresponds to a vector of size n, projecting out the list and then attempting to reestablish that it corresponds to a vector of size n never fails (and gets back to the same value). This explains the directedness of partial connections: we have, for all index n, im forget n $\lesssim_K$ List $\mathbb{N}$.

This machinery enables us to account for first-order connections between dependent and simple datatypes (*e.g.* Vec $\mathbb{N}$ n $\lesssim_K$ List $\mathbb{N}$) by composition of a partial connection (*e.g.* im forget n $\lesssim_K$ List $\mathbb{N}$) and a type equivalence (*e.g.* Vec $\mathbb{N}$ n $\simeq$ im forget n).

To account for higher-order transformations, we must generalize partial connections to relate any *flat partial order* through a pair of *monotone* functions subject to a similar ad-

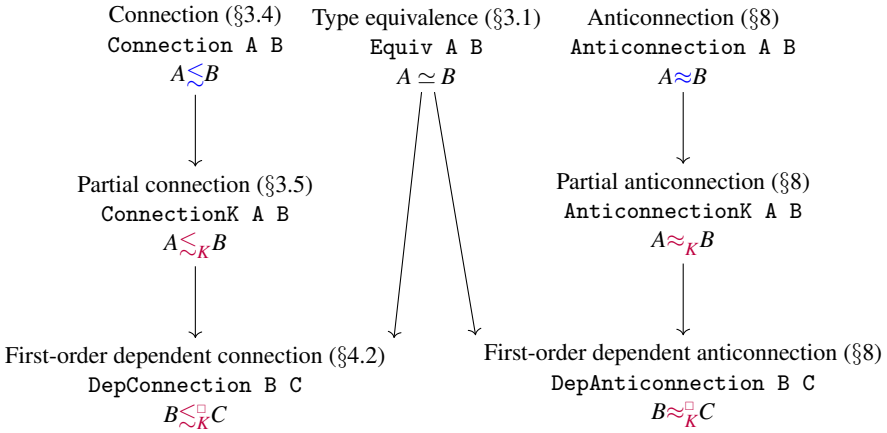


Fig. 1. Bird-eye view of the dependent interoperability framework

junction property. We thus obtain a type-theoretic version of monotone Galois connections (Section 3.4), or *connection* for short, written `A` $\lesssim$ `B`.

Monotone Galois connections, which are asymmetric, admit a symmetric treatment in the form of anticonnections (Section 8). As we will see, these variants enable us to sacrifice the completeness of the induced runtime coercions: an *anticonnection*, written `A` $\approx$ `B`, gives the freedom to arbitrarily abort a coercion—for example, if it turns out to be too costly to run. Similarly, a *partial anticonnection*, written `A` $\approx_K$ `B` allows us to relate a subset type that represents only a strict subset of its corresponding simple type—the translation from subset to simple types may thus be partial. In Section 3 and Section 4, we follow the above blueprint and introduce the necessary definitions, focusing on monotone connections. In Section 5, we further build higher-order connections to interoperate between functions that manipulate such structures. In Section 8, we extend our framework to support anticonnections. The overall view of the framework and its conceptual dependencies is summarized in Figure 1. To use the framework, the user instantiates some first-order dependent connections or anticonnections between pairs of dependent datatypes and simply-typed data structures of interest. Using the definitions of connections and anticonnections provided by the library, these first-order definitions are then automatically expanded to coercions on any higher-order function of interest.

3 Type-theoretic Partial Galois Connections

Intuitively, dependent interoperability is about exploiting an embedding from indexed types to simple types. This section formally captures such a relation, which we call *partial* because, as illustrated previously, some runtime errors might occur when crossing boundaries. To do so, we recall the usual model of partial functions in type theory (Section 3.3). We also translate the classical notion of a Galois connection in type-theoretical terms (Section 3.4). Our construction is inspired by the type-theoretic definition of an equivalence (Section 3.1), being particularly careful with the interplay between sets, propositions and truncations (Section 3.2). Specializing type-theoretic Galois connections to the model of partial functions yields the desired notion of partial connection (Section 3.5). Type equiva-

lences and partial connections underpin the definition of dependent connections (Section 4) whereas type-theoretic Galois connections provide the theoretical footing for transforming higher-order programs (Section 5).

We use Coq as both a formalization vehicle and an implementation platform. We make extensive use of *type classes* (Wadler & Blott, 1989) in order to define abstract structures and their properties, as well as relations among types. For instance, a dependent connection is a type class, whose instances must be declared in order to establish a connection between specific types, such as `Vec ℕ n` and `List ℕ`. As opposed to Haskell, type classes in Coq (Sozeau & Oury, 2008) can express arbitrary properties that need to be proven when declaring instances—for example, the monad type class in Coq *defines and imposes* the monad laws on each instance.

3.1 Type Equivalence

The notion of type equivalence offers a conceptual framework in which to reason about the relationships between types. A type equivalence between two types `A` and `B` is defined by a function `f: A → B` such that there exists a function `e_inv: B → A`, with proofs that it is both its left and right inverse together with a compatibility condition between these two proofs (Univalent Foundations Program, 2013). This definition plays a central role in Homotopy Type Theory (HoTT), as it is at the heart of the univalence axiom.

In this article, we exploit type equivalence as a means to (constructively) state that two types "are the same". In Coq:[4]

```
Class IsEquiv {A B: Type} (f: A → B) := {
 e_inv: B → A ;
 e_sect: e_inv ∘ f == id;
 e_retr: f ∘ e_inv == id;
 e_adj: ∀ x: A, e_retr (f x) = ap f (e_sect x)
}.
```

The properties `e_sect` and `e_retr` express that `e_inv` is the inverse of `f`. The definitions use the identity function `id`, point-wise equality between functions

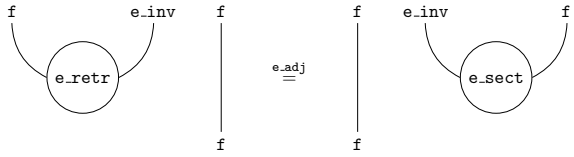
```
Notation "f == g" := (∀ x, f x = g x)
```

and the functorial action of `f`, transporting an equality between `x` and `y` to an equality between `f x` and `f y`:

```
Definition ap {A B: Type} (f: A → B) {x y: A} : x = y → f x = f y.
```

The extra coherence condition `e_adj` ensures that the equivalence is uniquely determined by the function `f`, that is, being an equivalence is proof-irrelevant. Diagrammatically, an equivalence is depicted by the following string diagram:

[4] Adapted from:
`http://hott.github.io/HoTT/coqdoc-html/HoTT.Overture.html#IsEquiv`



By virtue of being a proposition (thanks to `e_adj`), the fact that a function is an equivalence can be used in programs without having to keep track of which instance of `IsEquiv` was used since they are all equal. Failing that, a function using a given proof that `f` is an equivalence could not be shown to be propositionally equal to the *same* function using a syntactically distinct instance of `IsEquiv f`: the choice of proof would suddenly be relevant and pollute reasoning about programs.

The above class characterizes a *specific function* as witnessing an equivalence between two types, thereby allowing different functions to be thus qualified. Following the Coq HoTT library, we define the `Equiv` record type to specify that *there exists* an equivalence between two types `A` and `B`, denoted `A ≃ B`. The record thus encapsulates the equivalence function `e_fun` and defines a type relation:[5]

```
Record Equiv (A B: Type) := {
  e_fun: A → B ;
  e_isequiv: IsEquiv e_fun
}.
Notation "A ≃ B" := (Equiv A B)
```

Given an equivalence, we call the embedded functions *(total) coercions*, since they will be used to translate objects of type `A` to objects of type `B`, and conversely.

A corollary of the fact that being an equivalence is propositional gives us a simple way to prove that two equivalences e_1, e_2: `A ≃ B` are equal: it is sufficient to show that `e_fun` e_1 is equal to `e_fun` e_2. Note that this would not be the case for a mere isomorphism, which is essentially an equivalence without the coherence condition.

3.2 Homotopical interlude: HProp/HSet, Prop/Set & truncation

One of the initial goal of the sort `Prop` of Coq was to capture proof irrelevant properties that can be erased during extraction. This *syntactic* characterization is sometimes inadequate. On the one hand, it is too restrictive because some properties are proof irrelevant but cannot be typed in `Prop` (Mishra-Linger & Sheard, 2008). For example, Coq requires the class `IsEquiv` (Section 3.1) to be in `Type` because it packages a function. However, we can prove that any two instances of `IsEquiv f` are in fact equal: semantically, `IsEquiv` is proof-irrelevant.

On the other hand, there are elements of `Prop` that cannot be proven to be proof irrelevant. The most famous example of such an element is the equality type. Indeed, for every type `A`: `Type` and elements `a`,`b`: `A`, we have `a=b`: `Prop` in Coq, but proving that `a=b`

[5] Records differ from type classes in that they are not involved in (implicit) instance resolution; other than that, type classes are essentially records (Sozeau & Oury, 2008).

is irrelevant is independent from the theory of Coq as it corresponds to the Uniqueness of Identity Proofs (UIP) axiom (Hofmann & Streicher, 1994).

Therefore, we face two possible design choices. We could consider propositions in `Prop` and datatypes in `Set`, assuming UIP—which could be seen as controversial. Instead of relying on an axiom, we choose to require proof irrelevance *semantically* whenever it is needed. This semantic characterization of types with a proof-irrelevant equality is specified by the type class `IsHProp` as introduced in the Coq HoTT library:

```
Class IsHProp (T: Type) := is_hprop: ∀ x y: T, x = y.
```

In the same way, types that semantically form sets can be characterized by the type class `IsHSet`:

```
Class IsHSet A := is_hset:> ∀ a b: A, IsHProp (a = b).
```

The `:>` notation in the field `is_hset` declares an implicit coercion from inhabitants of the class `IsHSet A` to witnesses of the property `∀ a b: A, IsHProp (a = b)`.

Then, `HProp` is a record of a type `T: Type` for which there exists an instance of `IsHProp T` (and similarly for `HSet`).

```
Record HProp := hprop {
  _typeP:> Type;
  _isHProp: IsHProp _typeP
}.
```

The argument `_IsHProp` of the `hprop` constructor is declared as implicit because it can be inferred by the type class system. The implicit coercion `_typeP :> Type` allows to consider any inhabitant of `HProp` as a `Type`. For convenience, we have explicitly named the record constructor, `hprop`, shorter than the default that Coq would generate (`Build_HProp`).

We postulate the existence of a *propositional truncation* type (Awodey & Bauer, 2004) `Trunc : Type → Type`, in the form defined in the HoTT book (Univalent Foundations Program, 2013) and following its implementation in the HoTT Coq library (Bauer *et al.*, 2017). We exploit this mechanism below to define a proof-irrelevant error message type, `info_str`, while preserving consistency and compatibility with univalence. Specifically, we choose to use an error message in the form of a `string` (but we could also use more complete type information).

```
Definition info_str := hprop (Trunc string).
```

3.3 Modeling Partiality

As illustrated in Section 1, lifting values from simple to indexed types can fail at runtime. Thus, the type coercions we are interested in are *partial*. To denote—and reason about—partial functions, we resolve to use pointed sets (Hyland, 1991). In Coq, those are naturally modeled by working in the Kleisli category of the `option` monad, with a `None` constructor to indicate failure, and a `Some` constructor to indicate success.

Here, we also want to collect an error message to keep track of the coercion failure. Therefore, we define a `TError` monad (which may appear frightening at first sight):

```
Inductive TError A (info: HProp) :=
 | Some: A → TError A info
 | Fail: info → TError A info.
```

The `TError` monad is parametrized by the type of the error message, `info`. Note that we need the error message to be computationally irrelevant; this is why `info` has type `HProp`. As mentioned in Section 3.2, we use strings for error messages, defined in a type `info_str`.

The possibility to create an irrelevant string from a string is given by the `_with` function, for instance:

```
Example err: TError 𝔹 info_str := Fail (_with "coercion to 𝔹").
```

To lighten notations, we use the harpoon ⇀ to denote a (partial) function in the `TError` monad with `info_str`:

```
Notation "A ⇀ B" := (A → TError B info_str).
```

The `TError` monad is characterized by its identity `creturn`, binder `cbind` and lifting from pure functions to partial ones.[6]

```
Definition creturn {A} : A ⇀ A := Some.

Definition cbind A B {info}:
 TError A info → (A → TError B info) → TError B info :=
 fun a f ⇒ match a with
        Some a ⇒ f a
       | Fail i ⇒ Fail i
       end.
Notation "x ← e1 ; e2" := (cbind _ _ e1 (fun x ⇒ e2))

Definition clift A B: (A → B) → (A ⇀ B) :=
 fun f a ⇒ creturn (f a).
```

We use the traditional do-notation. For instance, function composition in the corresponding Kleisli category, denoted $\circ_K$, is defined as follows:

```
Definition kleisliComp {A B C: Type}: (A ⇀ B) → (B ⇀ C) → (A ⇀ C) :=
 fun f g a ⇒ b ← f a ; g b.
Notation "g ∘K f" := (kleisliComp f g)
```

We observe that the `TError` monad induces a flat partial order. The notion of partial order with a least element is naturally defined in Coq with the following type class:

```
Class IsPartialOrder⊥ (A: Type) := {
  rel: A → A → HProp where "x ≼ y" := (rel x y);
  ⊥: A;
```

[6] In Coq, parameters within curly braces are implicitly resolved.

```
  rel_refl: ∀ x, x ≼ x;
  rel_trans: ∀ x y z, x ≼ y → y ≼ z → x ≼ z;
  rel_antisym: ∀ x y, x ≼ y → y ≼ x → x = y;
  ⊥_is_least: ∀ x, ⊥ ≼ x;
}.
```

The `TError` monad on `HSet` induces a flat partial order that corresponds to equality on success values, and considers `Fail` as the least element of the ordering. More precisely: [7]

```
Instance IsPartialOrderTError (A: HSet): IsPartialOrder⊥ (TError A
    info_str) :=
 {| rel := fun a a' ⇒ @hprop (match a with
           | Some _ ⇒ a = a'
           | Fail _ ⇒ True
           end) _;
   ⊥ := Fail (_with "⊥") |}.
```

Besides, any partial order on the codomain of two functions gives us a way to compare these functions pointwise:[8]

```
Instance IsPartialOrder⊥_fun (A: Type)(B: A → Type)
  `{∀ a, IsPartialOrder⊥ (B a)}: IsPartialOrder⊥ (∀ a, B a) :=
 {| rel := fun f g ⇒ hprop (∀ a, f a ≼ g a);
   ⊥ := fun a ⇒ ⊥ |}.
```

3.4 Type-theoretic Galois connections

We now generalize the notion of type equivalence from types equipped with an equality to types equipped with a partial order. To witness such a relation, we work with *monotonic* functions, *i.e.* functions that preserve the partial order relation (and the least element).

```
Record monotone_function A B `{IsPartialOrder⊥ A} `{IsPartialOrder⊥ B} :=
 Build_Mon {
   f_ord:> A → B ;
   mon: ∀ x y, x ≼ y → f_ord x ≼ f_ord y;
   p_mon: f_ord ⊥ ≼ ⊥
 }.
Notation "A ⟶ B" := (monotone_function A B)
```

Monotonicity is expressed through the functorial action `f.(mon)`, thus following and generalizing the functorial action `ap f` of type equivalences (Section 3.1). We use a type class definition `Functor` to overload the notation `ap` of functorial action. The `:>` notation in the field `f_ord` declares an implicit coercion from `A ⟶ B` to `A → B`: we can transparently manipulate monotone functions as standard functions.

[7] Instance definitions require proving the properties declared in the instantiated class. These proofs have been elided here but can be found in the Coq scripts.

[8] In Coq, back-quoted parameters are nameless.

We now capture the notion of type connection between two partially ordered types A and B. The definition of IsConnection follows the general schema of a monotone Galois connection:

```
Class IsConnection {A B} `{IsPartialOrder⊥ A} `{IsPartialOrder⊥ B}
    (f: A ⟶ B) := {
  c_inv: B ⟶ A;
  c_sect: id ⪯ c_inv ∘ f ;
  c_retr: f ∘ c_inv ⪯ id;
}.
```

Note that there is no need to add an extra coherence c_adj—as it is the case for equivalences—because partial orders are in HProp and therefore the coherence is always satisfied. Moreover, using the fact that partial orders are antisymmetric in a strict sense (*i.e.*. rel_antisym yields a propositional equality), we can show that the inverse function e_inv of IsConnection f is in fact unique: being a connection is a mere proposition.

```
Instance IsConnectionIsHProp A B
     `{IsPartialOrder⊥ A}`{IsPartialOrder⊥ B}(f: A ⟶ B):
      IsHProp (IsConnection f).
```

Like for equivalences, we package type connections as a Coq type class in order to take advantage of the automatic instance resolution mechanism:

```
Class Connection A B `{IsPartialOrder⊥ A} `{IsPartialOrder⊥ B} := {
  c_fun: A ⟶ B ;
  c_isconn:> IsConnection c_fun
}.
Notation "A ≲ B" := (Connection A B)
```

3.5 Partial connections

We now intend to reconcile the general notion of type equivalence with the potential for errors, as modeled by the TError monad. The partial order over TError types yields the expected notion of connection in the Kleisli (bi-)category defined in Section 3.3. Composition amounts to monadic composition $\circ_K$, and identity to monadic identity creturn. We substantiate this intuition by specifying a *partial connection* to be:

```
Class IsConnectionK {A B: HSet} (f: A ⇀ B) := {
  pc_inv: B ⇀ A;
  pc_sect: creturn ⪯ pc_inv ∘_K f;
  pc_retr: f ∘_K pc_inv ⪯ creturn ;
}.
```

Because of the partial order we consider here, the property pc_sect reflects the fact that A embeds totally into B, *i.e.* c_inv ∘ f is in fact total and amounts to an identity function. In other words, IsConnectionK denotes a *Galois insertion*.

As a sanity check, we can prove that lifting an equivalence yields a partial connection in the Kleisli category.

```
Definition EquivToConnectionK (A B: HSet) (f: A → B) :
 IsEquiv f → IsConnectionK (clift f).
```

Finally, similarly to type equivalences and type connections, we expose partial connections into a type class:

```
Record ConnectionK (A B: HSet) := {
 pc_fun: A ⇀ B ;
 pc_isconn: IsConnectionK pc_fun
}.
Notation "A ≲_K B" := (ConnectionK A B)
```

The functions embedded in a partial connection are called *partial* coercions.

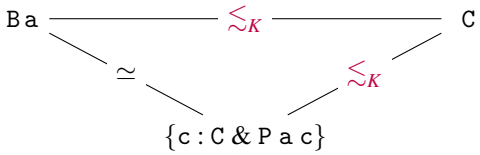
4 Partial Galois Connections for Dependent Interoperability

We now exploit partial connections to setup a verified framework for dependent interoperability. In this context, we are specifically interested in partial connections between indexed types, such as `Vec` $\mathbb{N}$, and simple types, such as `List` $\mathbb{N}$. We call this kind of partial connection a *dependent connection* (Section 4.3). Beside treating the general case, we also identify special cases where establishing a dependent connection is significantly easier, such as when the indexed type encodes a decidable property or when its index is an `HSet` (Section 4.4). In these cases, the user is freed from having to prove tedious coherence properties.

As explained at the beginning of Section 3, a major insight of this work is that a dependent connection can be defined by composition of two different kinds of type relations, using as intermediary the subset type that characterizes the *image* of the indexed type through its total embedding to the simple type. More specifically, we compose:

- a *total* equivalence between indexed types and subset types of the form $\{$`c: C & P c`$\}$ whose logical content `P` is carefully quarantined from their computational content `C` (Section 4.1);
- a *partial* connection between subset types and simple types, in the Kleisli category of the `TError` monad (Section 4.2).

This strategy amounts to the following commuting diagram, for a given index `a`:



The resulting dependent connection is therefore also a partial connection, defined in the Kleisli category. For instance, to establish the connection between vectors and lists, we exploit the property that characterizes points in the image of the total embedding of `Vec` $\mathbb{N}$ `n` into `List` $\mathbb{N}$, namely $\exists$ `v: Vec` $\mathbb{N}$ `n, forget n v = l`, which is logically equivalent to the property `length l = n`, and hence use the subset type $\{$`l: List` $\mathbb{N}$ & `length l = n`$\}$. Intuitively, this subset captures the *meaning* of the index of `Vec` $\mathbb{N}$.

In the diagram above, and the rest of this article, we adopt the following convention: the type index is `A`: `Type`, the type family is `B`: `A` → `Type`, the plain type is `C`: `Type`, and the logical proposition capturing the meaning of the index is `P`: `A` → `C` → `Type`.

4.1 Equivalence Between Indexed and Subset Types

The first step is a total equivalence between indexed types and subset types. In our dependent interoperability framework, this is the only relation that the programmer has to manually establish and prove. For instance, to relate lists and vectors, the programmer must establish that, for a given `n`:

$$\texttt{Vec}\ \mathbb{N}\ \texttt{n} \simeq \{\ \texttt{l}\colon \texttt{List}\ \mathbb{N}\ \&\ \texttt{length}\ \texttt{l} = \texttt{n}\ \}$$

Recall from Section 3.1 that establishing this total equivalence requires providing two functions:

```
Vec_to_list n: Vec ℕ n → { l: List ℕ & length l = n }
list_to_Vec n: { l: List ℕ & length l = n } → Vec ℕ n
```

These functions capture the computational content of the conversion between the two data structures.[9] The programmer must also prove that they are inverse of each other. In addition, she needs to prove the `e_adj` property. This coherence property is generally quite involved to prove; fortunately, there are two canonical ways to establish it. The first one is to use a function that transforms any isomorphism into an equivalence (a process called *adjointification*). The second option is to use the fact that if the equivalence is between two sets (that is, inhabitants of `HSet`), then the coherence is automatic. This raises the question of how to prove that a type is a set.

Remark: Proving that a type is a set. In HoTT, the primitive way prove that a type is a set is to use Hedberg's theorem (Univalent Foundations Program, 2013), which states that every type with a decidable equality is actually a set. Thus, to prove that a type A is a set, it is enough to provide a term of type $\forall\ (\texttt{x}\ \texttt{y} : \texttt{A}),\ (\texttt{x} = \texttt{y}) + \texttt{not}\ (\texttt{x} = \texttt{y})$. This is typically done for inductive types by pattern matching on constructors, as for instance for boolean:

```
Definition Decidable_eq_𝔹 (x y : 𝔹) : (x = y) + not (x = y) :=
 match x,y with
  true, true  ⇒ inl eq_refl
 | false, false ⇒ inl eq_refl
 | true, false  ⇒ inr absurd_eq_𝔹
 | false, true  ⇒ inr (absurd_eq_𝔹 ∘ inverse)
 end.
```

where `absurd_eq_𝔹` is a proof that `not (true = false)`.

[9] Note that a programmer may very well choose to define a conversion that reverses the elements of the structure. As long as the equivalence is formally proven, this is permitted by the framework; any lifting that requires the equivalence uses the user-defined canonical instance.

Then, once instances of `IsHSet` have been provided for some types, other instances can be inferred using the fact that dependent sums preserve sets and that a dependent product is a set as soon as its codomain is, as given by:

```
Instance IsHSet_∀ P (Q : P → Type) `{HQ : ∀ x, IsHSet (Q x)}: IsHSet (∀ x, Q x).
```

4.2 Connection Between Subset and Simple Types

The second relation we exploit is a *partial connection* between subset types and simple types such as, for a given `n`:

$$\{\ \texttt{l}\colon \texttt{List}\ \mathbb{N}\ \&\ \texttt{length l} = \texttt{n}\ \} \lesssim_K \texttt{List}\ \mathbb{N}$$

Obviously, going from the subset type to the simple type never fails, as it is just the first projection of the underlying dependent pair π_1. However, the other direction is not always possible: it depends if the given `n` is equal to the length of the considered list.

We follow the approach of Tanter and Tabareau (2015) for coercing values of any simple type `C` to values of a subset type $\{\texttt{c}\colon \texttt{C}\ \&\ \texttt{P c}\}$, for any *decidable* predicate `P`. A proposition `A` is an instance of the `Decidable` type class when there exists a function that returns either a proof of `A` or a proof of `not A`.

```
Class Decidable (A: HProp) := dec: A + (not A).
```

Here, we restrict the use of the `Decidable` type class to `HProp` to force the element computed by the decidability function to be irrelevant. Note that in practice, this often means using explicit truncation (`hprop (Trunc ...)`), just as we did for the error message information in `TError` (Section 3.2).

Using decidability, Tanter and Tabareau perform the type coercion through a runtime check, relying on an axiom to capture the potential for coercion failure. In the monadic setting we adopt here, there is no need for axioms anymore; their technique amounts to establishing a partial connection $\{\texttt{c}\colon \texttt{C}\ \&\ \texttt{P c}\} \lesssim_K \texttt{C}$. We capture this partial connection between subset types and simple types with the following instance:

```
Definition Decidable_ConnectionK (C:HSet) P `{∀ c, Decidable (P c)}:
 {c:C & P c} ≲_K C := {| pc_fun := clift π₁ : {c:C & P c} ⇀ C |}.
```

The embedding function `pc_fun` is the (lifting of the) first projection function π_1 (the type ascription is necessary to help the Coq type inference algorithm). The inverse function is the `to_subset` function below, which is essentially a monadic adaptation of the coercion operator of Tanter and Tabareau (2015):

```
Definition to_subset {C: HSet} {P} `{∀ c, Decidable (P c)}: C ⇀ ({c: C & P c})
 := fun c ⇒
    match dec (P c) with
    | inl p ⇒ Some (c;p)
    | inr _ ⇒ Fail (_with ("subset conversion"))
    end.
```

Note that proof irrelevance of $\texttt{P c}$ is crucial, because when going from the subset type $\{\texttt{c}\colon \texttt{C}\ \&\ \texttt{P c}\}$ to the simple type `C` and back, the element of $\texttt{P c}$ is inferred by the decision

procedure, and there is no reason for it to be the same as the initial element of `P c`. By proof-irrelevance, both proofs are considered equal, and the partial connection is established.

4.3 Connection Between Dependent and Simple Types

We now define a partial connection between dependent and simple types by composing the type equivalence and partial connection described above. The *dependent connection* class below captures the necessary requirements for two structures to interoperate. `DepConnection` corresponds to *first order* dependent connections, in as much as it only relates data structures. In Section 5, we shall develop higher-order dependent connections, which enable us to operate over functions. As explained above, we define a class so as to piggy-back on instance resolution to find the canonical connections automatically.

```
Class DepConnection (A: HSet) (B: A → HSet) (C: HSet) := {
  P: A → C → HProp;
  total_equiv:> ∀ a, B a ≃ {c: C & P a c};
  partial_equiv:> ∀ a, {c: C & P a c} ≲_K C;
  f_ca: C ⇀ A;
  P_f_ca: ∀ a (b: B a), f_ca ∘_K ((partial_equiv a).(pc_fun))
                    ((total_equiv a).(e_fun) b) = Some a;
}.
Notation "B ≲^□_K C" := (DepConnection _ B C)
```

(The index type `A` can always be inferred from the context so the notation $\lesssim^{\square}_K$ omits it.) A key ingredient to establishing a dependent connection between the type family `B` and the simple type `C` is the property `P` that connects the two relations. Note that the partial connection and total equivalence with the subset type are lifted to point-wise connections, *i.e.* they must hold for all indices `a`.[10]

The `DepConnection` class also includes an index synthesis function, f_{ca}, which recovers a canonical index from a data of simple type. In the case of `List` $\mathbb{N}$, it is always possible to compute its length, but as we will see in the case of stack machine instructions (Section 6), synthesizing an index may fail. The f_{ca} function is used for defining higher-order connections, *i.e.* for automatically lifting functions (Section 5). The property $P_{f_{ca}}$ states that if we have a value `c: C` that was obtained through the coercion from a value of type `B a`, then f_{ca} is defined on `c` and recovers the original index `a`.

Note that `P` need not be decidable. For example, the following datatype `mlist` of *masked lists* (which forbids certain elements recorded in a blacklist from appearing in the data structure)

```
Inductive In: A → list A → Type :=
| here: ∀ a xs, In a (cons a xs)
| there: ∀ a b xs, In a xs → In a (cons b xs).

Inductive mlist: list A → Type :=
```

[10] To define a dependent connection, Coq must also be able to infer that the type `C` is an `HSet`.

```
| nil: ∀ l, mlist l
| cons: ∀ x l, not (In x l) → mlist l → mlist l.
```

admits a dependent connection to plain lists, through an undecidable predicate P stating that every elements of the plain lists do not belong to the blacklist, since equality of the list's elements itself may not be decidable. In the (many) other cases where P is in fact decidable, we provide a simpler entry-point to obtain such a connection, as shown in Section 4.4.

Finally, for all index a, there is a partial connection between B a and C:[11]

```
Definition DepConnection_ConnectionK (A: HSet) (B: A → HSet)
    (C: HSet) `{B ≲□_K C} (a: A): B a ≲_K C :=
 {| pc_fun := to_simpl;
    pc_isconn := {| pc_inv := to_dep a|} |}.
```

The functions used to establish the partial connection are `to_simpl`, which is the standard composition of the coercion functions `pc_fun` ∘ `e_fun`, and the function `to_dep`, which is the Kleisli composition of the inverse functions, (`clift e_inv`) $\circ_K$ `pc_inv`.

4.4 Simplifying the Definition of a Dependent Connection

In practice, requiring the simple type C to be an HSet allows to alleviate the burden on the user, because some coherences become automatically satisfied. We define a function `DepConnection_hset` that exploits this and establishes a dependent connection without requiring the extra coherences `e_adj`.

Additionally, note that the `DepConnection` class is independent of the particular partial connection between the subset type and the simple type. Therefore, we provide a smart constructor for dependent connections, applicable whenever the partial connection with the subset type is given by a decidable property:

```
DepConnection_hset (A: HSet) (B: A → HSet) (C: HSet) P
 `{∀ a c, Decidable (P a c)}
 (f_bc: ∀ a, B a → {c: C & P a c}) (f_cb: ∀ a, {c: C & P a c} → B a) (f_ca: C ⇀ A):
 (∀ a, (f_cb a) ∘ (f_bc a) == id ) → (∀ a, (f_bc a) ∘ (f_cb a) == id) →
 (∀ a (b: B a), f_ca (f_bc a b).1 = Some a) → B ≲□_K C.
```

Using `DepConnection_hset`, establishing a new dependent connection between two types such as `Vec` ℕ and `List` ℕ boils down to providing a decidable predicate, two inverse coercion functions, and the index synthesis function (`length`). The programmer must then prove three equations corresponding to the properties of the coercion functions and that of the index synthesis function.

Frequently, the decidable predicate merely states that the synthesized index is equal to the proposed index (*i.e.* P := `fun a c` ⇒ (`clift length`) c = `Some a`). We provide another convenient instance constructor `DepConnection_eq`, specialized for this situation. Declaring the canonical dependent connection between `Vec` ℕ and `List` ℕ amounts to:

[11] We skip over the (unsurprising) proofs of the section `pc_sect` and retraction `pc_retr` properties associated to `pc_inv` by postponing them thanks to the `Program` mode of Coq.

```
Instance DepEquiv_Vec_list_ℕ: Vec ℕ ≲□K list ℕ :=
 DepConnection_eq (Vec ℕ) (list ℕ) (clift length)
        Vec_to_list list_to_Vec.
```

Recall that the index synthesis function may be partial: for instance, synthesizing the index of a presumably balanced binary tree must fail if the simply-typed tree provided is in fact unbalanced.

5 Higher-Order Galois Connections

Having defined first-order dependent connections, which relate indexed types (*e.g.* `Vec ℕ`) and simple types (*e.g.* `List ℕ`), we now turn to *higher-order* dependent connections. These connections relate partial functions over simple types, such as `List ℕ ⇀ List ℕ`, to partially equivalent functions over indexed types, such as `∀ n, Vec ℕ (n + 1) ⇀ Vec ℕ n`.

Higher-order dependent connections support the application scenarios of dependent interoperability described in Section 1. Importantly, while programmers are expected to define their own first-order dependent connections, higher-order dependent connections are automatically derived based on the available canonical first-order dependent connections.

5.1 Defining A Higher-Order Dependent Connection

Consider that two first-order dependent connections `B₁ ≲□K C₁` and `B₂ ≲□K C₂` have been previously established. We can construct a Galois connection between functions of type `∀ a: A, B₁ a ⇀ B₂ a` and functions of type `C₁ ⇀ C₂`:

```
Instance HOConnection {A: HSet} {B₁ B₂: A → HSet} {C₁ C₂: HSet}:
 (B₁ ≲□K C₁) → (B₂ ≲□K C₂) → (∀ a, B₁ a ⇀ B₂ a) ≲ (C₁ ⇀ C₂)
 := fun _ _ ⇒
    {| c_fun := Build_Mon
          (fun f ⇒ to_simpl_dom (fun a b ⇒ x ← f a b; to_simpl x)) _ _ ;
       c_isconn := {| c_inv := Build_Mon
          (fun f a b ⇒ x ← to_dep_dom f a b; to_dep _ x) _ _ |}|}.
```

The definition of the `HOConnection` instance relies on two auxiliary functions, `to_dep_dom` and `to_simpl_dom`.

`to_dep_dom` lifts a function of type `C ⇀ D` for any type `D` to an equivalent function of type `∀ a. B a ⇀ D`. It simply precomposes the function to lift with `to_simpl` in the Kleisli category:

```
Definition to_dep_dom {A D: HSet} {B: A → HSet} {C: HSet}
  `{B ≲□K C}(f: C ⇀ D) (a: A): B a ⇀ D := f ∘K to_simpl.
```

`to_simpl_dom` lifts the domain of a function in the other direction. Its definition is more subtle because it requires computing the index `a` associated to `c` before applying `to_dep`. This is precisely the *raison d'être* of the f_{ca} function provided by the `DepConnection` type class (Section 4.3).

```
Definition to_simpl_dom {A D: HSet} {B: A → HSet} {C: HSet}
```

```
    ‘{B ≲□_K C} (f: ∀ a: A, B a ⇀ D): C ⇀ D := fun c ⇒
  a ← a_f_ca c;
  b ← to_dep a c ;
  f a b.
```

Crucially, the proof that `HOConnection` is a proper connection is done once and for all. This is an important asset for programmers because the proof is quite technical. It implies proving equalities in the Kleisli category and requires in particular the extra property $P_{f_{ca}}$. We come back to proof techniques in Section 7.

As the coherence condition of `HOConnection` involves equality between functions, the proof makes use of the functional extensionality axiom, which states that `f == g` is equivalent to `f = g` for any dependent functions `f` and `g`. This axiom is very common and compatible with both UIP and univalence, but it cannot be proven in Coq for the moment, because equality is defined as an inductive type, and the dependent product is of a coinductive nature (Boulier *et al.*, 2017).

In order to cope with pure functions of type $\forall$ `a`, B_1 `a` $\to$ B_2 `a`, we first embed the pure function into the monadic setting and then apply a dependent connection:

```
Definition lift {A: HSet} {B₁ B₂: A →HSet}{C₁ C₂: HSet}
  ‘{(∀ a, B₁ a ⇀ B₂ a) ≲ (C₁ ⇀ C₂)} :
  (∀ a, B₁ a → B₂ a) → C₁ ⇀ C₂ :=
 fun f ⇒ c_fun (fun a b ⇒ creturn (f a b)).
```

This definition is straightforward, yet it provides a convenient interface to the user of the dependent interoperability framework. For instance, lifting the function:

```
Vector.map: ∀ (f: ℕ → ℕ) (n: ℕ), Vec ℕ n → Vec ℕ n
```

is a mere `lift` away:

```
Definition map_list (f: ℕ → ℕ): list ℕ ⇀ list ℕ := lift (Vector.map f).
```

Note that it is however not (yet) possible to lift the tail function `Vector.tl`: $\forall$ `n`, `Vec` $\mathbb{N}$ (`S n`) $\to$ `Vec` $\mathbb{N}$ `n` because there is no dependent connection between `fun n ⇒ Vec` $\mathbb{N}$ (`S n`) and `List` $\mathbb{N}$. Fortunately, the framework is extensible and we will see in the next section how to deal with this example, among others.

5.2 A Library of Higher-Order Dependent Connections

`HOConnection` is but one instance of an extensible library of higher-order dependent connection classes. One of the benefits of our approach to dependent interoperability is the flexibility of the framework. Automation of higher-order dependent connections is open-ended and user-extensible. We now discuss some useful variants, which provide a generic skeleton that can be tailored and extended to suit specific needs.

Index injections. `HOConnection` only covers the pointwise application of a type index over the domain and codomain types. This fails to take advantage of full-spectrum dependent types: a type-level function could perfectly be applied to the type index. For instance, if we want to lift the tail function `Vector.tl`: $\forall$ `n`, `Vec` $\mathbb{N}$ (`S n`) $\to$ `Vec` $\mathbb{N}$ `n` to a function of

type `List` $\mathbb{N} \to$ `List` $\mathbb{N}$, then the domain index is obtained from the index `n` by application of the successor function.

Of particular interest is the case where the index function is an inductive constructor. Indeed, inductive families are commonly defined by case analysis over some underlying inductive type (Brady *et al.*, 2004). Semantically, we characterize constructors through their defining characteristic: they are *injective*. We thus define a class of injections where the inverse function is allowed to fail:

```
Class IsInjective {A B: HSet} (f: A → B) := {
  i_inv: B ⇀ A;
  i_sect: i_inv ∘ f == creturn ;
  i_retr: (clift f) ∘_K i_inv ≼ creturn
}.
```

We can then define a general instance of `DepConnection` that captures the use of an injection on the index. Note that for the sake of generality, the domain of the injection can be a different index type `A'` from the one taken by `B`:

```
Instance Connection_Inj (A A': HSet)(B: A → HSet)(C: HSet)
 (f: A' → A) '{IsInjective _ _ f}
 '{B ≲□_K C}: (fun a ⇒ B (f a)) ≲□_K C
```

This new instance now makes it possible to lift the tail function from vectors to lists:

```
Definition pop: list ℕ ⇀ list ℕ :=
  lift (Vector.tl: ∀ n:ℕ, @Vec ℕ DecidablePaths_ℕ (S n) → Vec ℕ n).
```

As expected, when applied to the empty list, the function `pop` returns `Fail`.

In the other direction, we use the `unlift` operator

```
Definition unlift {A: HSet}{B₁ B₂: A → HSet}{C₁ C₂: HSet}
 '{(∀ a, B₁ a ⇀ B₂ a) ≲ (C₁ ⇀ C₂)} :
 (C₁ → C₂) → ∀ a, B₁ a ⇀ B₂ a
```

to transform a `pop` function on lists to the dependent type ∀ `n`, `Vec` ℕ (`S n`) ⇀ `Vec` ℕ `n`.

```
Definition vpop: ∀ n: ℕ, Vec ℕ (S n) ⇀ Vec ℕ n :=
  unlift (B₁ := n : ℕ, Vec ℕ (S n)) (List.tl (A := ℕ)).
```

This function can only be applied to a non-empty vector, but if it does not return a vector of the input length reduced by one, a coercion error is reported.

Composing connections. With curried dependently-typed functions, the index of an argument can be used as an index of a subsequent argument (and return type), for instance:

∀ `a a'`, $\mathtt{B}_1$ `a` → $\mathtt{B}_2$ `a a'` ⇀ $\mathtt{B}_3$ `a a'`

We can define an instance of ≲ to form a new connection on ∀ `a a'`, $\mathtt{B}_1$ `a` → $\mathtt{B}_2$ `a a'` ⇀ $\mathtt{B}_3$ `a a'` from a connection on ∀ `a'`, $\mathtt{B}_2$ `a a'` ⇀ $\mathtt{B}_3$ `a a'`, for a fixed `a`, provided that we have established that $\mathtt{B}_1$ $\lesssim^{\square}_K$ $\mathtt{C}_1$:

```
Instance HOConnection₂
 (A A': HSet) (B₁: A → HSet) (B₂ B₃: A → A' → HSet) (C₁ C₂ C₃: HSet) :
 (B₁ ≲□_K C₁) → (∀ a, ((∀ a': A', B₂ a a' ⇀ B₃ a a') ≲ (C₂ ⇀ C₃))) →
 (∀ a a', B₁ a → B₂ a a' ⇀ B₃ a a') ≲ (C₁ → C₂ ⇀ C₃).
```

The definition of the two functions involved in the composed connection is obtained by using the coercion functions provided by the connection on $\forall$ a', B$_2$ a a' $\rightharpoonup$ B$_3$ a a' and B$_1$ $\lesssim^{\square}_K$ C$_1$ accordingly, as we have done for HOConnection. For instance, the function from left to right is given by:

```
Definition HOConnection₂_fun
 {A A': HSet} {B₁: A → HSet} {B₂ B₃: A → A' → HSet} {C₁ C₂ C₃: HSet} :
 (B₁ ≲□_K C₁) → (∀ a, ((∀ a': A', B₂ a a' ⇀ B₃ a a') ≲ (C₂ ⇀ C₃))) →
 (∀ a a', B₁ a  B₂ a a' ⇀ B₃ a a') → C₁ → C₂ ⇀ C₃ :=
 fun _ _ f c₁ c₂ ⇒ to_simpl_dom (fun a b₁ ⇒ c_fun (fun a' ⇒ f a a' b₁) c₂) c₁.
```

The main benefit of defining such an instance is that the proof that the two above functions form a connection is done once and for all and can be found automatically by type class inference. We do not dive into the technical details of this instance, but it is crucial to handle the stack machine example of Section 1: as will be explained in Section 6, this mechanism allows us to handle the following function exec': $\forall$ n m, dstack n → dinstr n m → dstack m which is composed of two dependent connections, one on instructions and one on stacks.

Index dependencies. It is sometimes necessary to reorder arguments in order to be able to compose connections, accounting for (functional) dependencies between indices. This reordering of parameters can be automatized by defining an instance that tries to flip arguments to find a potential connection:

```
Instance HOConnection₂_sym
  (A A': HSet) (B₁ B₂ B₃: A → A' → HSet) {C₁ C₂ C₃: HSet}
 `{ (∀ a a', B₂ a a' → B₁ a a' ⇀ B₃ a a') ≲ (C₂ → C₁ ⇀ C₃) }:
  (∀ a a', B₁ a a' → B₂ a a' ⇀ B₃ a a') ≲ (C₁ → C₂ ⇀ C₃)
```

Note that this instance has to be given a very low priority (omitted here) because it should be used as a last resort, otherwise one would introduce cycles during type class resolution. The stack machine example in the following section also exploits this instance, which enables us to transform the function exec: $\forall$ n m, dinstr n m → dstack n → dstack m into exec': $\forall$ n m, dstack n → dinstr n m → dstack m, thus triggering further transformations.

6 A Certified, Interoperable Stack Machine

To demonstrate our approach, we address the shortcomings of extraction identified in Section 1 and present a certified yet interoperable interpreter for a toy stack machine. Let us recall the specification of the interpreter:

exec: $\forall$ n m, dinstr n m → dstack n → dstack m

This definition enforces—by construction—an invariant relating the size of the input stack and output stack, based on which instruction is to be executed.

In the simply-typed setting, we would like to offer the following interface:

```
safe_exec: instr → List ℕ → List ℕ
```

while dynamically enforcing the same invariants (and rejecting ill-formed inputs).

This example touches upon two challenges. First, it involves two connections, one dealing with instructions and the other dealing with stacks. Once those connections have been defined by the user, we shall make sure that our machinery automatically finds them to lift the function `exec`. Second, and more importantly, type indices flow in a non-trivial manner through the type signature of `exec`. For instance, providing an empty stack means that we must forbid the use of the `IPlus` instruction. Put otherwise, the lifting of the dependent instruction depends on the (successful) lifting of the dependent stack. As we shall see, the index `n` is (uniquely) determined by the input list size while the index `m` is (uniquely) determined by `n` and the instruction being executed. In effect, the automation of higher-order lifting compiles a dependent type into a suitably ordered sequence of index computations and verifications.

In this process, users are only asked to provide the first-order type equivalences specific to their target domain, `dstack` $\lesssim_K^{\square}$ `List` ℕ and ∀ `n`, (`dinstr n` $\lesssim_K^{\square}$ `instr`). Using these instances, the role of our framework is threefold: (1) to linearize the indexing flow, through potentially reordering function arguments; (2) to identify the suitable first-order equivalences, through user and library provided instances; (3) to propagate the indices computed through dynamic checks, through the constructive reading of the higher-order connections.

6.1 From Stacks to Lists

As hinted at in Section 1, the type of `dstack` cannot be properly extracted to a simply-typed system. Indeed, it is defined by large elimination over natural numbers (*i.e.* by pattern-matching over a natural number – a term – to produce a type) and there is therefore no natural, simply-typed data structure to extract it to. As a result, extraction in Coq resorts to unsafe type coercions (Letouzey, 2004). However, using specific domain knowledge, the programmer can craft a connection with a list, along the lines of the connection between vectors and lists. We therefore (constructively) witness the following subset equivalence:

```
dstack n ≃ {l: List ℕ & clift length l = Some n}
```

by which we map size-indexed tuples to lists.[12] Crucially, this transformation involves a change of representation: we move from tuples to lists. For our framework to automatically handle this transformation, we merely need to register the suitable dependent connection by means of an instance declaration:

```
Instance Connection_dstack: dstack ≲_K^□ list ℕ :=
```

[12] Note that the equality `clift length l = Some n` is equivalent to the simpler `length l = n`, but the framework is tailored to encompass potential failure. This could be avoided by defining a more specific function than `DepEquiv_eq` for the case where computation of the index never fails.

```
DepConnection_eq dstack (list ℕ) (clift length)
        dstack_to_list list_to_dstack.
```

The definition of this dependent connection is very similar in nature to the one already described between vectors and lists, so we refer the reader to the implementation for details.

6.2 From Indexed Instructions to Simple Instructions

The interoperable version of indexed instructions is more natural to construct: indexed instructions are a standard inductive family whose indices play a purely logical role.

```
Inductive dinstr: ℕ → ℕ → Type :=
| IConst: ∀ n, ℕ → dinstr n (S n)
| IPlus: ∀ n, dinstr (S (S n)) (S n).
```

Merely erasing this information gives an inductive type of (simple) instructions:

```
Inductive instr: Type :=
| NConst: ℕ → instr
| NPlus: instr.
```

Nonetheless, relating the indexed and simple version is conceptually more involved. Indeed, the first index cannot be guessed from the simply-typed representation alone: the size of the input stack must be provided by some other means. Knowing the size of the input stack, we can determine the expected size of the output stack for a given simple instruction:

```
Definition instr_index n (i: instr): TError ℕ _ :=
 match i with
  | NConst _ ⇒ Some (S n)
  | NPlus ⇒ match n with
          | S (S n) ⇒ Some (S n)
          | _ ⇒ Fail (_with "invalid instruction")
        end
 end.
```

The dependent connection is thus *parameterized* by the input size `n` and only the output size `m` needs to be determined from the simple instruction:

```
∀ n, dinstr n m ≃ {i: instr & instr_index n i = Some m}.
```

Once again, we inform the system of this new connection through a suitable instance declaration:

```
Instance Connection_instr n: dinstr n ≲□K instr :=
 DepConnection_eq (dinstr n) instr (instr_index n) (dinstr_to_instr n) (
    instr_to_dinstr n).
```

6.3 Lifting the Interpreter

Having specified our domain-specific connections, we are left to initiate the instance resolution so as to *automatically* obtain the desired coercion of the interpreter `exec`. To do so, we simply appeal to the `lift2` operator provided by the framework, which is similar to `lift` from Section 5.1 save for the fact that it deals with two-index types:

```
lift2 A A' (B1 B2 B3: A → A' → HSet) C1 C2 C3
 `{∀ a a', B1 a a' → B2 a a' ⇁ B3 a a' ≲K C1 → C2 ⇁ C3}:
  (∀ a a', B1 a a' → B2 a a' → B3 a a') → C1 → C2 ⇁ C3.
```

The definition of `simple_exec` is then:

```
Definition simple_exec: instr → list ℕ ⇁ list ℕ := lift2 exec.
```

`lift2` matches upon the skeleton of our dependent function `exec`, lifts it to a monadic setting and triggers the instance resolution mechanism of Coq. This (single) command is enough to build the term `simple_exec` with the desired type, together with the formal guarantee that it is valid with respect to the dependent program we started from.

6.4 Diving into the Generated Term

Printing the generated term (by telling Coq to show the connection instances) is instructive:

```
simple_exec = lift2
 (HOConnection2_sym
   (HOConnection2 Connection_dstack
     ( a: ℕ, HOConnection (Connection_instr a) Connection_dstack))) exec
```

We witness three generic transformations, provided by the framework, of the function: `HOConnection2_sym`, which has reordered the input arguments so as to first determine the size of the input stack; `HOConnection2`, which has made the size of the input list available to subsequent transformations; and `HOConnection`, which has transferred the size of the output list as computed from the simple instruction to the output stack.

Now, when printing `simple_exec` by telling Coq to unfold definitions, we recover the description of a function in monadic style that actually performs the expected computation:

```
simple_exec =  fun (i: instr) (l: List ℕ) ⇒
(* lift l to a dstack ds of size [(length l)] *)
  ds ← (c' ← to_subset l; Some (list_to_dstack c'));
(* compute the index associated to [(length l)] for i;
   this may fail depending on the instruction *)
  m ← instr_index (length l) i;
(* lift i to a dependent instruction di *)
  di ← (c' ← to_subset i; Some (instr_to_dinstr (length l) m c'));
(* perform exec (note the reverse order of di and ds)
   and convert the result to a list *)
  Some (dstack_to_list (exec (length l) m di ds)) .1
```

6.5 Extraction to OCaml

In Section 3.3, we introduced the `TError` monad as *essentially* the option monad. For practical purposes, the failure constructor of `TError` takes additional arguments for producing informative error messages: we capture the type we are trying to coerce to and a message to help diagnose the source of the error.

More importantly, having defined a custom error monad enables us to tailor program extraction when targeting an impure language. In an impure language like OCaml, it is indeed possible—and strongly advised—to write in direct style, using runtime exceptions to implement the `TError` monad. The success constructor of the monad is simply erased, and its failure constructor is projected to a runtime exception (*e.g.* `failwith` in OCaml).

```
Extract Inductive TError ⇒
(* Transparent extraction of TError:
   - if Some t, then extract plain t
   - if Fail, then fail with a runtime coercion exception *)
  "" [ "" "(let f s = failwith
                (String.concat """" (""Coercion failure: ""]@
                  (List.map (String.make 1) s))) in f)" ]
  "(let new_pattern some none = some in new_pattern)".
```

This allows us to avoid affecting the consistency of the host language Coq —conversely to Tanter and Tabareau (2015), we do not introduce inconsistent axioms to represent coercion errors—while preserving the software engineering benefits of not imposing a monadic framework on external components.

We can now revisit the interaction with the extracted function:

```
# simple_exec NPlus [1;2];;
: int list = [3]
# simple_exec NPlus [];;
Exception: (Failure "Coercion failure: invalid instruction").
```

and confirm that an invalid application of `simple_exec` does not yield a segmentation fault, but an informative exception.

7 A Homotopical Detour

We now briefly reflect on the proof techniques we used to build the verified dependent interoperability framework and implement the different examples.

Many of the proofs of sections and retractions, either on general instances (Sections 4 and 5) or on domain-specific connections (as we shall see below), require complex reasoning on equality. This means that particular attention must be paid to the definition of conversion functions. In particular, the manipulation of equality must be done through explicit rewriting using the transport map (which is the predicative version of `ap` introduced in Section 3):

```
Definition transport {A: Type} (P: A → Type) {x y: A}
  (p: x = y) (u: P x): P y :=
```

```
 match p with eq_refl ⇒ u end.
Notation "p # x" := (transport _ p x)
```

Transport is trivially implemented by path induction (*i.e.* pattern-matching the proof of equality against `eq_refl` or, put otherwise, performing induction on equality), but making explicit use of transport is one of the most important technical insights brought by the advent of HoTT. It is crucial as it enables to encapsulate and reason abstractly on rewriting, without fighting against dependent types.[13] Indeed, although equality is *presented* through an inductive type in Coq, it remains best dealt with through abstract rewriting—a lesson that was already familiar to observational type theorists (Altenkirch *et al.*, 2007). The reason is that it is extremely difficult to prove equality of two pattern matching definitions by solely reasoning by pattern matching. Conversely, it is perfectly manageable to prove equality of two different transportations.

For instance, the definition of `instr_to_dinstr` must be defined by pattern matching on the instruction and transport (comments express the specific type of the goal in each branch of pattern matching):

```
Definition instr_to_dinstr n n':
 {i: instr & instr_index n i = Some m} → dinstr n n' := fun x ⇒
match x with (i;v) ⇒ (match i with
(* ⊢ Some (S n) = Some n' → dinstr n n' *)
 | NConst k ⇒ fun v ⇒ Some_inj v # IConst k
 | NPlus    ⇒ match n with
 (* ⊢ None = Some n' → dinstr 0 n' *)
    0 ⇒ fun v ⇒ None_is_not_Some v
 (* ⊢ None = Some n' → dinstr 1 n' *)
   | S 0 ⇒ fun v ⇒ None_is_not_Some v
 (* ⊢ Some (S n) = Some n' → dinstr (S (S n)) n' *)
   | S (S n) ⇒ fun v ⇒ Some_inj v # IPlus
end end) v end.
```

where `None_is_not_Some` is a proof that `None` is different from `Some a` for any `a` (in the sense that `None = Some a` implies anything) and `Some_inj` is a proof of injectivity of the constructor `Some`.

The benefit of using the encapsulation of rewriting through transport is that now, we can independently prove auxiliary lemmas on transport (such as `Some_inj` and `None_is_not_Some`) and use them in the proof. For instance, we can state how transport behaves on the `IConst` instruction by path induction:

```
Definition transport_instr_Const (n m k: ℕ) (e: S n = m) :
  dinstr_to_instr _ _ (e # (IConst k)) = (NConst k; ap Some e).
```

and similarly for `IPlus`. Armed with these properties on transport, we can then prove the retraction of `dinstr m n` $\simeq$ `{i: instr & instr_index n i = Some m}` by pattern match-

[13] A fight that Coq usually announces with "`Abstracting over the terms...`" and wins by declaring "`is ill-typed.`"

ing on instructions and integers, together with some groupoid laws (@ is equality concatenation and path_sigma is a proof that equalities of the projections imply equality of dependent pairs).

```
Definition DepEquiv_instr_retr n m
  (x: {i: instr & instr_index n i = Some m}):
  (dinstr_to_instr n m) ∘ (instr_to_dinstr n m) x = x :=
 match x with (i;v) ⇒ (match i with
(*  ⊢  Some (S n) = Some m →
      dinstr_to_instr n m (Some_inj v # IConst n0) = (NConst n0; v) *)
 NConst k ⇒ fun v ⇒
      transport_instr_Const @
      path_sigma eq_refl (is_hprop _ _)
| NPlus ⇒ match n with
  (*  ⊢  None = Some m →
        dinstr_to_instr 0 m (Fail_is_not_Some v) = (Nplus; v) *)
    | 0 ⇒ fun v ⇒ None_is_not_Some v
  (*  ⊢  None = Some m →
        dinstr_to_instr 1 m (Fail_is_not_Some v) = (Nplus; v) *)
    | S 0 ⇒ fun v ⇒ None_is_not_Some v
  (*  ⊢  Some (S n) = Some m →
        dinstr_to_instr (S (S n)) m (Some_inj v # IPlus) = (NPlus; v) *)
    | S (S n) ⇒ fun v ⇒
      transport_instr_Plus @
      path_sigma eq_refl (is_hprop _ _))
   end end) v end.
```

We believe that this new way of proving equalities—initially introduced to manage higher equalities in syntactical homotopy theory—is very promising for proving equalities on definitions done by pattern matching and thus proving properties on dependent types.

Remark: Using the convoy pattern (Chlipala, 2013) and with extreme care, we could have written the function instr_to_dinstr directly, without explicitly appealing to transport:

```
Definition instr_to_dinstr' n n'
  (iv: {i: instr & valid_instr i n n'}): dinstr n n' :=
 match iv with
 | (i; v) ⇒
  match i return (valid_instr i n n' → dinstr n n') with
  | NConst n0 ⇒
   fun v ⇒
    let e := Some_inj v in
    match e in (_ = m)
       return (valid_instr (NConst n0) n m → dinstr n m)
    with
    | eq_refl ⇒ fun _ ⇒ IConst n0
    end v
```

```
| NPlus ⇒ (* ... *)
```

However, the retraction proof is *stated* using this very definition: to prove the lemma `DepEquiv_instr_retr`, we will have to manually unfold the convoy patterns (with four more occurrences in the `NPlus` case) in the proof. For example, in the `NConst n0` case, we will have to show that the proof `e` computed by `Some_inj v` is indeed `eq_refl`, thus allowing us to simplify the term and extract the return value `IConst n0`. While it was (painfully) manageable to implement `instr_to_dinstr` directly, doing a direct proof manipulating it is excessively cumbersome. The overhead imposed by the direct approach thus forbids tackling anything but trivial programs whereas the transport-based approach is compositional and modular.

This should come at no surprise to implementors of pattern-matching compilers (McBride, 2000; Sozeau, 2010; Cockx *et al.*, 2016) who had, time and again, to fight a struggle between generalization of dependent types – which introduces equalities – and simplification of the resulting terms – which collapses as many equalities as possible. These difficulties motivated in a large part the adoption of UIP for it provides further opportunities for simplification thus making the resulting terms manageable.

8 Dependent Interoperability with Anticonnections

The type-theoretic Galois connections `IsConnection` and `IsConnectionK` are *monotone*: in particular, this means that their sections (`c_sect`, `pc_sect`) must be *greater* than the identity:

```
Class IsConnection ... := {
 ...
 c_sect: id ≼ c_inv ∘ f
 ...
}
```

```
Class IsConnectionK ... := {
 ...
 pc_sect: creturn ≼ pc_inv ∘_K f
 ...
}
```

In other words, the section of a connection is not allowed to fail. In previous work (Dagand *et al.*, 2016), our definitions were weaker:[14]

```
Class IsAnticonnection ... := {
 ...
 ac_sect: c_inv ∘ f ≼ id
 ...
}
```

```
Class IsAnticonnectionK ... := {
 ...
 apc_sect: pc_inv ∘_K f ≼ creturn
 ...
}
```

This design allows the coercions to fail in both directions: we call such an object an anticonnection.

Despite their resemblance, an anticonnection is *not* an antitone Galois connection: it still acts monotonically on objects, unlike an antitone connection. By relaxing the property

[14] We have applied some renaming to the definitions in (Dagand *et al.*, 2016) to match the ones of this article.

`c_sect` into `ac_sect`, we lose the unicity of the inverse: unlike monotone connections where it is unique when it exists, there may be many inverse functions `c_inv` verifying the required properties of an anticonnection. It is to be expected, as we may implement various degrees of approximation. However, it has the consequence of making the choice of anticonnection proof-relevant: reasoning about programs lifted through an anticonnection will involve reasoning about which approximations were applied.

Alternatively, we could have dropped the section property `c_sect` altogether from the definition of a Galois connection. Such a Galois preconnection is nonetheless weaker than our notion of anticonnection: specialized to partial functions, a preconnection would leave unspecified the behavior of the inverse image off the range of `f` (such as producing a dummy value rather than failing) whereas an anticonnection forbids the inverse image to produce values out of thin air.

As it turns out, the choice of (monotone) partial connection *vs.* partial anticonnection represents an interesting design tradeoff for dependent interoperability, which was not envisioned in our prior work.

On the one hand, the monotone connection forces the conversion from simple types to indexed types to be exact, therefore the intermediate subset type *must be equivalent* to the image restriction of the embedding of the indexed type. This is a strong property, which amounts to the completeness of the predicate defining the subset type: the resulting coercion succeeds for every valid simply-typed value.

It also means that the monotone framework captures a notion of *type precision*, akin to that used in recent accounts of gradual typing (Siek *et al.*, 2015; Garcia *et al.*, 2016) to characterize the amount of static information that a type carries. This suggests that building a form of gradual typing on top of the dependent interoperability framework would be better served by the monotone framework.

On the other hand, the anticonnection allows the predicate to be a *conservative approximation* of the exact image subset type, thus sacrificing completeness (while remaining sound). This allows programmers to use decision procedures that may conservatively fail when the property actually holds. While a degenerate sound approximation that always fails is obviously useless, there is a range of interesting intermediate approximations that allow a tradeoff between completeness and either efficiency or ease of certification.

Unlike monotone connections, an anticonnection yields a symmetric and transitive relation between types. This allows establishing a connection between two indexed types where none is "lesser" in any sense than the other. For example, we would be at a loss to establish a monotone connection between "integers vectors of size `m`" and "lists whose values are less than `k`". However, we can easily establish an anticonnection between these two types, plain lists of integers serving as a common ground.

In the following, we first present the definition of a `Checkable` type class that captures the requirement of the decidable and sound approximation of a property, and its integration in our framework. We end with a brief illustration.

8.1 Sound Approximation of Properties

Instead of imposing actual *decidability*, the `Checkable` type class only asks for the predicate to be *checkable*, *i.e.* there must exist a decidable, sound approximation. We also demand proof irrelevance of `P` (via `HProp`).

```
Class Checkable (A: HProp) := {
  check: HProp;
  check_dec: Decidable check ;
  convert: check → A
}.
```

Of course, every decidable property is checkable.

The approximate decision procedure may be conservative and return `false` when `P c` does in fact hold, but it has to be the case that if it returns `true`, then `P c` holds.

We can use `Checkable` to extend the `to_subset` function defined in Section 4.2, so that it now applies the sound approximation decision procedure for the embedded logical proposition:

```
Definition to_subset {C: HSet} {P} `{∀ c, Checkable (P c)}: C ⇀ ({c:C & P c})
 := fun c ⇒
    match dec (check (P c)) with
    | inl p ⇒ Some (c; convert p)
    | inr _ ⇒ Fail (_with ("subset conversion"))
    end.
```

If `to_subset` succeeds, the proof of success of the approximation is converted (by using the implication `convert` from the `Checkable` instance) to a proof of the property. Otherwise an error is raised.

We can use `to_subset` to prove that every subset of a checkable property is in partial anticonnection (noted $\approx_K$) with the underlying type.

```
Definition Checkable_AnticonnectionK (C:HSet) P `{∀ c, Checkable (P c)}:
 {c:C & P c} ≈_K C :=
 {| apc_fun := clift π1 : {c:C & P c} ⇀ C |}.
```

Finally, note that there is no point in using `Checkable` for monotone connections: the uniqueness of the adjoint in the monotone setting forces the approximation to be both sound and complete; in other words, it forces the predicate to be `Decidable`. Using the anticonnection framework (with `Checkable`) does slightly affect some proofs in our framework, but does not otherwise affect the statement of the main theorems and functions.

8.2 Relation between connections and anticonnections

In general, connections and anticonnections are not comparable. However, when looking at partial connections, we can show that any monotone partial connection `A` $\lesssim_K$ `B` gives rise to a partial anticonnection `A` $\approx_K$ `B`. In the other direction, we can show that any partial anticonnection `A` $\approx_K$ `B` gives rise to a monotone partial connection providing a proof of the missing section, namely

```
Definition Anticonnection_Connection (A B: HSet) (H : A ≈_K B):
      creturn ≼ (apc_inv (apc_fun H)) ∘_K (apc_fun H) → A ≲_K B.
```

Using this fact, plus the fact that every decidable property is checkable, we can factorize the framework as follows. All the framework of Sections 3 and 4 is defined for checkable properties in the anticonnection case. Then the computational content and part of the properties are directly inherited for decidable properties in the monotone framework. It just remains to prove the missing section of monotone partial connections. For instance, as explained above, the function `to_subset` automatically works for decidable properties and the proof that every subset of a decidable property is in *monotone* partial connection with the underlying type only requires to prove that

```
∀ (c:{c : C & P c}), Some c = to_subset (π_1 c).
```

8.3 Anticonnections in Action

To illustrate the use of anticonnections, let us consider another indexed type `listZ b`, for lists of natural numbers indexed by a Boolean `b` that indicates whether the list contains the element 0:

```
Inductive listZ : 𝔹 → Type :=
| nilZ : listZ false
| cons_zero : ∀ (n:ℕ) b, n = 0 → listZ b → listZ true
| cons_pos : ∀ (n:ℕ) b, not (n = 0) → listZ b → listZ b.
```

One specific interest of this indexed type is that a `product` function can use the fact that the index is true to immediately return zero, instead of recursively computing the actual product.

```
Definition product (b: 𝔹) (l: listZ b): ℕ :=
 if b then 0
 else product' _ l. (* recursive product *)
```

Establishing a dependent connection between `listZ` and `list ℕ` requires stating the property that captures the semantics of the boolean index:

```
Definition listZ_P (b:𝔹) (l:list ℕ) : HProp := hprop (contains ℕ 0 l = b).
```

Note `listZ_P` is a mere proposition because $\mathbb{B}$ is a set. Here, we exploit the fact that the property that captures the semantics of the boolean index is decidable and can thus been defined using a function into booleans. This way of reflecting a property using an equality on booleans is done typically in SSReflect (Gonthier & Mahbouhi, 2010).

Alternative characterizations of the property not using a decision procedure (using for instance $\exists$ `n` : $\mathbb{N}$, `nth_error l n = Some 0`) can also be used, but they may require the use of the truncation operator (as described in Section 3.2) to ensure that we have a mere proposition. Focusing on the proof irrelevance of a type rather than requiring its decidability is more in the spirit of homotopy type theory.

The conversion functions from `list ℕ` to `listZ` and back are straightforward and omitted here.

```
Definition listZ_to_list (b: 𝔹) (l:listZ b) : {l : list ℕ & listZ_P b l}.
Definition list_to_listZ (b: 𝔹): {l : list ℕ & listZ_P b l} → listZ b.
```

A `Decidable` instance for `listZ_P` is automatically inferred (because equality on booleans is decidable), but note that it implies a *linear* cost for checking. If we want to instead use a *constant* time checking approach, we can exploit `Checkable` to define a more efficient, yet *incomplete*, decision procedure that only checks the k first elements of a list. The approximative property `listZ_P_bound` is defined as follows:

```
Definition listZ_P_bound (b:𝔹) (k :ℕ) (l:list ℕ) : HProp :=
 if b then hprop (contains_bound k 0 l = true) else hprop False.
```

If one needs to assert `listZ true`, and 0 is found in the first k elements, the property holds; otherwise, it conservatively does not hold. Of course, the list might in fact contain 0 at a later position, so the index-based optimization of `product` will not be taken advantage of in such cases. Conversely, if one needs to assert `listZ false`, meaning that the list does *not* contain 0, the approximative decision procedure always fail. (A more clever definition would exploit the length of the list: if the list has less than k elements, it can examine them all, otherwise, it needs to conservatively fail.)

We then show that `listZ_P_bound` is decidable, and use it to define an instance of the `Checkable` typeclass for `listZ_P`:

```
Instance listZ_P_checkable bound b l : Checkable (listZ_P b l) :=
 {| check := listZ_P_bound b bound l |}.
```

Finally, we can define the dependent connection

```
Instance DepEquiv_listZ_list (bound : ℕ) : listZ ≈ᵏ□ list ℕ.
```

Note that in addition to efficiency, using `Checkable` can also be helpful to bypass undecidability. For instance, if we admit *infinite* lists, then exhaustively checking their elements is simply undecidable; a conservative examination of a fixed prefix is.

9 Related Work

As far as we know, the term *dependent interoperability* was originally coined by Osera *et al.* (2012) as a particularly challenging case of *multi-language semantics* between a dependently-typed and a simply-typed language. The concept of multi-language semantics was initially introduced by Findler and Felleisen (2007) to capture the interactions between a simply-typed calculus and a uni-typed calculus (where all closed terms have the same unique type).

Our approach is strictly more general in that we make no assumption on the dependent types we support: as long as the user provides Galois connections, our framework is able to exploit them automatically. In particular, we do not require a one-to-one correspondence between constructors: the connection is established at the type level, giving the user the freedom to implement potentially partial transformations. We also account for more general connections through partial index synthesis functions; Osera *et al.* (2012) assume that these functions are total and manually introduced by users. Finally, while their work is fundamentally grounded in a syntactic treatment of interoperability, ours takes its roots in

a semantic treatment of Galois connections internalized in Coq. We are thus able to give a presentation from first principles while providing an executable toolbox in the form of a Coq library that is entirely verified.

Dynamic typing with dependent types. Dependent interoperability can also be considered within a single language, as explored by Ou *et al.* (2004). The authors developed a core language with dependent function types and subset types augmented with three special commands: `simple{e}`, to denote that expression `e` is simply well-typed, `dependent{e}`, to denote that the type checker should statically check all dependent constraints in `e`, and `assert(e, T)` to check at runtime that `e` produces a value of (possibly-dependent) type `T`. The semantics of the source language is given by translation to an internal language relying, when needed, on runtime-checked type coercions.

However, dependent types are restricted to refinement types where the refinements are pure Boolean expressions, as in the hybrid typing work of Knowles and Flanagan (2010). This means that the authors do not address the issues related to indexed types, including that of providing correct marshaling functions between representations, which is a core challenge of dependent interoperability.

Coercions for subset types. Tanter and Tabareau (2015) also explore the interaction between simple types and refinements types in a richer setting than Ou *et al.* (2004): their approach is developed in Coq, and thus refinements are any proposition (not just Boolean procedures), and they accommodate explicitly proven propositions. They support safe coercions between simple types and subset types by embedding runtime checks to ensure that the logical component of a subset type is satisfied. Our notion of dependent connection builds upon the idea of coercion to subset types—we use subset types as mediators between simple types and indexed types. But instead of using an inconsistent axiom in the computational fragment of the framework to represent coercion errors, we operate within a `TError` monad (recall that we do use a fairly standard axiom, functional extensionality, in the non-computational fragment of the framework). Imposing a monadic style augments the cost of using our framework within Coq, but we can recover the convenience of non-monadic signatures upon extraction. Finally, just like Ou *et al.* (2004), the work of Tanter and Tabareau (2015) only deals with subset types and hence does not touch upon dependent interoperability in its generality.

The fact that dependent connections only abstractly rely on coercions to subset types should make it possible to derive instances for other predicates than the `Checkable` ones. For instance, in the setting of the Mathematical Components library using the SSreflect proof language, properties are better described through Boolean reflection (Gonthier & Mahbouhi, 2010). Using Boolean functions is very similar to using decidable/checkable properties, so that framework should provide a lot of new interesting instances of partial connections between subset and simple types in the Kleisli category of the `TError` monad.

Gradual typing. Multi-language semantics are directly related to *gradual typing* (Siek & Taha, 2006), generalized to denote the integration of type disciplines of different strengths within the same language. This relativistic view of gradual typing has already been explored in the literature for disciplines like effect typing (Bañados *et al.*, 2014; Toro &

Tanter, 2015; Bañados Schwerter *et al.*, 2016) and security typing (Disney & Flanagan, 2011; Fennell & Thiemann, 2013) annotated type systems (Thiemann & Fennell, 2014), and more recently, refinement types (Lehmann & Tanter, 2017). Compared to previous work on gradual typing, dependent interoperability is challenging because the properties captured by type indices can be semantically complex. Also, because indices are strongly tied to specific constructors, coercion requires marshaling (Osera *et al.*, 2012). This contrasts with casts, which typically only check tags at first-order types and produce wrappers for function types.

In our framework, coercions must be explicitly summoned with uses of `lift`. However, as already observed by Tanter and Tabareau (2015), the implicit coercions of Coq[15] can be used to direct the type checker to automatically insert liftings when necessary, thus yielding a controlled form of gradual typing.

This being said, there is still work to be done to develop a full theory of gradual dependent types. It would be interesting to explore how the abstract interpretation foundations of gradual typing as a theory of dealing with type information of different levels of precision (Garcia *et al.*, 2016) can be connected with our framework for dependent connections, which relate types of different precision. As discussed in Section 8, there seems to be a direct relation between the notion of precision in gradual typing and the existence of a monotone partial Galois insertion. Further exploration of this relation is future work.

Ornaments. Our work is rooted in a strict separation of the computational and logical content of types, which resonates with the theory of ornaments (McBride, 2010), whose motto is "datatype = data structure + data logic". Ornaments have been designed as a construction kit for inductive families: while data structures—concrete representation over which computations are performed—are fairly generic, data-logics—which enforce program invariants—are extremely domain-specific and ought to be obtained on-the-fly, from an algebraic specification of the invariants.

In particular, two key constructions are algebraic ornaments (McBride, 2010) and relational ornaments (Ko & Gibbons, 2013). From an inductive type and an algebra (over an endofunctor on $\mathscr{S}et$ for algebraic ornaments, over an endofunctor on $\mathscr{R}el$ for relational ornaments), these ornaments construct an inductive family satisfying—by construction—the desired invariants. The validity of this construction is established through a type equivalence, which asserts that the inductive family is equivalent to the subset of the underlying type satisfying the algebraic predicate.

However, the present work differs from ornaments in several, significant ways. First, from a methodological standpoint: ornaments aim at creating a combinatorial toolbox for creating dependent types from simple ones. Following this design goal, ornaments provide correct-by-construction transformation of data structures: from a type and an algebra, one obtains a type family. In our framework, both the underlying type and the indexed type must pre-exist, we merely ask for a (constructive) Galois connection between them. Conceptually, partial connections subsume ornaments in the sense that an ornament automatically gives rise to a Galois connection. However, ornaments are restricted to inductive types,

[15] See (The Coq Development Team, 2016), Chapter 18.

while Galois connections offer a uniform framework to characterize the refinement of any type, including inductive families.

Functional ornaments. To remediate these limitations, the notion of functional ornaments (Dagand & McBride, 2012) was developed. As for ornaments, functional ornaments aim at transporting functions from simple, non-indexed types to more precise, indexed types. The canonical example consists in taking addition over natural numbers to concatenation of lists: both operations are strikingly similar and can indeed be described through ornamentation. Functional ornaments can thus be seen as a generalization of ornaments to first-order functions over inductive types.

So far, however, such generalization of ornaments have failed to carry over higher-order functions and genuinely support non-inductive types. The original treatment of functional ornaments followed a semantic approach, restricting functions to be catamorphisms and operating over their defining algebras. More recent treatment (Williams *et al.*, 2014), on the other hand, is strongly grounded in the syntax, leading to heuristics identifying common syntactic artefacts (such as the structure of a pattern-matching definition) that are difficult to rationalize semanticallly.

By focusing on type connections, our approach is conceptually simpler: our role is to consistently preserve type information, while functional ornaments must infer or create well-indexed types out of thin air. By restricting ourselves to checkable properties, we also afford the flexibility of runtime checks and, therefore, we can simply lift values to and from dependent types by merely converting between data representations. Finally, while the original work on functional ornaments used a reflective universe, we use type classes as an open-ended and extensible meta-programming framework. In particular, users are able to extend the framework at will, unlike the clients of a fixed reflective universe.

Refinement types. Our work shares some similarities with refinement types (Xi & Pfenning, 1998; Rondon *et al.*, 2008). Indeed, dependent connections are established through an intermediary type equivalence with user-provided subset types, which is but a type-theoretic incarnation of a refinement type. From refinement types, we follow the intuition that most program invariants can be attached to their underlying data structures. We thus take advantage of the relationship between simple and indexed types to generate runtime checks. The runtime checks associated to refinements differ from gradual refinement types (Lehmann & Tanter, 2017) in that here we support any decidable (or checkable) predicate, but do not support *imprecise* logical statements, which are the main novelty introduced by gradual refinement types. Additionally, unlike Sekiyama *et al.* (2015), our current prototype fails to take advantage of the algebraic nature of some predicates, thus leading to potentially inefficient runtime checks. In principle, this shortcoming could be addressed by integrating algebraic ornaments in the definition of type connections. Besides, instead of introducing another manifest contract calculus and painstakingly developing its meta-theory, we leverage full-spectrum dependent types to internalize the coercion machinery through type-theoretic Galois connections.

Such internalization of refinement techniques has permeated the interactive theorem proving community at large, with applications ranging from improving the efficiency of small-scale reflection (Cohen *et al.*, 2013), the step-wise refinement of specifications down

to correct-by-construction programs (Delaware *et al.*, 2015; Swierstra & Alpuim, 2016), or the transfer of theorems across type isomorphisms (Zimmermann & Herbelin, 2015). Our work differs from the first application because we are interested in *safe* execution outside of Coq rather than *fast* execution in the Coq reduction engine. It would nonetheless be interesting to attempt a similar parametric interpretation of our dependent connections. Our work also differs from step-wise refinements in the sense that we transform dependently-typed programs to and from simply-typed ones, while step-wise refinements are concerned with incrementally determining a relational specification. Our approach shares some similarity with the latter work, which takes advantage of type isomorphisms to transfer results across datatype/logical representations. To do so, the authors extend Coq with an ad-hoc plugin: it would be interesting to exploit such a plugin in our own work, thus simplifying and rationalizing our use of type classes.

Type isomorphisms & effects. The definition of type equivalence benefits from the extreme simplicity of the semantics of types: we work in the context of a pure and total programming language, equivalence of types amounts (morally) to equivalence of `HSet`s. Similarly, partial connections benefits from the relative simplicity afforded by the interpretation of types as pointed sets.

Handling richer effects remain an open question. This is particularly relevant when considering the interoperable extraction to OCaml, which features exceptions, memory cells and input/outputs. The extensive body of literature on type isomorphisms (Cosmo, 2005) may provide some useful guiding principles for generalizing type equivalence to these settings. In particular, Laurent (Laurent, 2005) has studied the impact of continuations on type isomorphisms in a linear setting (through a denotational model based on game semantics) while Clairambault (Clairambault, 2012) has exploited this denotational model to characterize type isomorphisms in a simply-typed language with sums, products and memory cells. These works rely on a carefully crafted model, enabling the computation of isomorphisms, which aligns with the idea of *characterizing* the type isomorphisms valid in the given languages. This is not strictly necessary in our setting, since the user can always establish an equivalence manually. Such systematic characterizations nonetheless provide a useful basis for an automated decision procedure.

More directly relevant to our work, Levy (Levy, 2017) offers a language-generic specification of type isomorphisms by means of *contextual isomorphisms*. This work provides a blueprint for adapting the notion of type isomorphism to a wide range of side-effects by segregating types between value types (strict, effect-free) and computation types (lazy, side-effecting). It would be interesting to understand how this distinction fares in our dependently-typed setting.

10 Conclusion

In this article, we have given a semantic account of dependent interoperability through the notion of type-theoretic Galois connections. Our definitions were set up to be directly mechanizable: this resulted in a library of generic connection-preserving program transformations and generic proofs of said connections. To our knowledge, this is the first im-

plementation of a dependent interoperability framework. Our verified Coq implementation includes all the examples presented in this article.

In the process, Coq has been a particularly relevant medium to study and develop dependent interoperability. We were led to take advantage of its dual nature, as a programming language and as a theorem prover. The fundamentally mathematical notion of partial Galois connection and its higher-order counterpart thus arose from a development (and refactoring) process driven by programming concerns. As a result, we were able to fully embed dependent interoperability in Coq itself, including the statements and proofs of correctness of the interoperability layer.

Our library rests crucially on type classes: (partial) connections are expressed as type classes, allowing users to plug and immediately play with their domain-specific connections. We also expose the coercion operators through this mechanism. In effect, lifting programs is implemented through a logic program, critically relying on higher-order unification. Type classes were instrumental in enabling this form of meta-programming.

Future work. As a first step, we wish to optimize the runtime checks compiled into the interoperability wrappers. Indeed, dependent types often exhibit a tightly-coupled flow of indexing information. Case in point is the certified stack machine, whose length of the input list gives away the first index of the typed instruction set while its second index is obtained from the raw instruction and the input length. By being systematic in our treatment of such dataflows, we hope to identify their sequential treatments and thus minimize the number of (re)computations from simply-typed values.

The similarities with standard dataflow analysis techniques are striking. Some dependent connections (typically, `DepEquiv_eq`) are genuine *definition sites*. For instance, the input list of the stack machine *defines* its associated index. Other connections are mere *use sites*. For instance, the first argument of the typed instruction cannot be determined from a raw instruction: one must obtain it from the input list. The second argument of the typed instruction can be computed from the first one, thus witnessing a *use-definition chain*. Conceptually, compiling a dependent type to a runtime check amounts to turning this dataflow graph into a sequential program.

Taking full advantage of this representation opens many avenues for generalizations. For instance, our current definition of dependent connection insists on being able to recover an index from a raw value through the mandatory $\mathtt{f}_{ca}\colon \mathtt{C} \rightharpoonup \mathtt{A}$. As such, this precludes the definition of many connections, such as the relation between natural numbers and finite sets (*i.e.* bounded natural numbers, the bound being unknown), or between raw lambda terms and intrinsic dependently-typed terms (the types being unknown and, *a priori*, not inferable).

Finally, perhaps inspired by the theory of ornaments, we would like to avoid marshaling values across inductive types and their algebraically ornamented families. Indeed, when converting from, say, lists to vectors, we perform a full traversal of the list to convert it to a vector that is, essentially, the same datatype. Most of our conversion functions are nothing but elaborate identity functions. However, the computational complexity of these conversions is likely to change the overall complexity of the algorithm. By taking advantage of this structural information and, perhaps, some knowledge of the extraction mechanism, we would like to remove this useless and inefficient indirection.

Acknowledgments

We thank Max New for important terminological clues, as well as the anonymous reviewers for helping us enhance the presentation of this work.

References

Altenkirch, Thorsten, McBride, Conor, & Swierstra, Wouter. (2007). Observational equality, now! *Pages 57–68 of: Proceedings of the ACM workshop on programming languages meets program verification (plpv 2007)*.

Awodey, Steven, & Bauer, Andrej. (2004). Propositions as [types]. *Journal of logic and computation*, **14**(4), 447–471.

Bañados, Felipe, Garcia, Ronald, & Tanter, Éric. (2014). A theory of gradual effect systems. *Pages 283–295 of: Proceedings of the 19th acm sigplan conference on functional programming (icfp 2014)*. Gothenburg, Sweden: ACM Press.

Bañados Schwerter, Felipe, Garcia, Ronald, & Tanter, Éric. (2016). Gradual type-and-effect systems. *Journal of functional programming*, **26**, 19:1–19:69.

Bauer, Andrej, Gross, Jason, Lumsdaine, Peter LeFanu, Shulman, Michael, Sozeau, Matthieu, & Spitters, Bas. (2017). The HoTT library: A formalization of homotopy type theory in Coq. *Pages 164–172 of: Proceedings of the 6th ACM SIGPLAN conference on certified programs and proofs (cpp 2017)*. Paris, France: ACM Press.

Boulier, Simon, Pédrot, Pierre-Marie, & Tabareau, Nicolas. (2017). The next 700 syntactical models of type theory. *Pages 182–194 of: Proceedings of the 6th ACM SIGPLAN conference on certified programs and proofs (cpp 2017)*. Paris, France: ACM Press.

Brady, Edwin, McBride, Conor, & McKinna, James. (2004). *Types for proofs and programs*. Lecture Notes in Computer Science, vol. 3085. Springer-Verlag. Chap. Inductive Families Need Not Store Their Indices, pages 115–129.

Chlipala, Adam. (2013). *Certified programming with dependent types*. MIT Press.

Clairambault, Pierre. (2012). Isomorphisms of types in the presence of higher-order references (extended version). *Logical methods in computer science*, **8**(3).

Cockx, Jesper, Devriese, Dominique, & Piessens, Frank. (2016). Eliminating dependent pattern matching without K. *Journal of functional programming*, **26**, e16.

Cohen, Cyril, Dénès, Maxime, & Mörtberg, Anders. (2013). Refinements for free! *Pages 147–162 of: Proceedings of the 3rd international conference on certified programs and proofs (cpp 2013)*.

Cosmo, Roberto Di. (2005). A short survey of isomorphisms of types. *Mathematical structures in computer science*, **15**(5), 825–838.

Dagand, Pierre-Évariste, & McBride, Connor. (2012). Transporting functions across ornaments. *Pages 103–114 of: Proceedings of the 17th acm sigplan conference on functional programming (icfp 2012)*. Copenhagen, Denmark: ACM Press.

Dagand, Pierre-Evariste, Tabareau, Nicolas, & Éric Tanter. (2016). Partial type equivalences for verified dependent interoperability. *Pages 298–310 of: Proceedings of the 21st acm sigplan conference on functional programming (icfp 2016)*. Nara, Japan: ACM Press.

Delaware, Benjamin, Pit-Claudel, Clément, Gross, Jason, & Chlipala, Adam. (2015). Fiat: Deductive synthesis of abstract data types in a proof assistant. *Pages 689–700 of: Proceedings of the 42nd ACM SIGPLAN-SIGACT symposium on principles of programming languages (popl 2015)*. Mumbai, India: ACM Press.

Disney, Tim, & Flanagan, Cormac. (2011). Gradual information flow typing. *International workshop on scripts to programs*.

Fennell, Luminous, & Thiemann, Peter. (2013). Gradual security typing with references. *Pages 224–239 of: Proceedings of the 26th computer security foundations symposium (csf)*.

Findler, Robert Bruce, & Felleisen, Matthias. (2002). Contracts for higher-order functions. *Pages 48–59 of: Proceedings of the 7th acm sigplan conference on functional programming (icfp 2002)*. Pittsburgh, PA, USA: ACM Press.

Garcia, Ronald, Clark, Alison M., & Tanter, Éric. (2016). Abstracting gradual typing. *Pages 429–442 of: Proceedings of the 43rd ACM SIGPLAN-SIGACT symposium on principles of programming languages (popl 2016)*. St Petersburg, FL, USA: ACM Press.

Gonthier, Georges, & Mahbouhi, Assia. (2010). An introduction to small scale reflection in Coq. *Journal of formalized reasoning*, **3**(2), 95–152.

Hofmann, Martin, & Streicher, Thomas. (1994). The groupoid model refutes uniqueness of identity proofs. *Pages 208–212 of: Proceedings of the 9th symposium on logic in computer science (lics '94)*. IEEE Computer Society Press.

Hyland, J. M. E. (1991). First steps in synthetic domain theory. *Pages 131–156 of: Proceedings of the international conference on category theory*. Como, Italy: Springer-Verlag.

Knowles, Kenneth, & Flanagan, Cormac. (2010). Hybrid type checking. *Acm transactions on programming languages and systems*, **32**(2), Article n.6.

Ko, Hsiang-Shang, & Gibbons, Jeremy. (2013). Relational algebraic ornaments. *Pages 37–48 of: Proceedings of the ACM SIGPLAN workshop on dependently typed programming (dtp 2013)*. ACM Press.

Laurent, Olivier. (2005). Classical isomorphisms of types. *Mathematical structures in computer science*, **15**(5), 969–1004.

Lehmann, Nico, & Tanter, Éric. (2017). Gradual refinement types. *Pages 775–788 of: Proceedings of the 44th ACM SIGPLAN-SIGACT symposium on principles of programming languages (popl 2017)*. Paris, France: ACM Press.

Letouzey, Pierre. (2004). *Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq*. Ph.D. thesis, Université Paris-Sud.

Levy, Paul Blain. (2017). Contextual isomorphisms. *Pages 400–414 of: Proceedings of the 44th ACM SIGPLAN-SIGACT symposium on principles of programming languages (popl 2017)*. Paris, France: ACM Press.

Matthews, Jacob, & Findler, Robert Bruce. (2007). Operational semantics for multi-language programs. *Pages 3–10 of: Proceedings of the 34th ACM SIGPLAN-SIGACT symposium on principles of programming languages (popl 2007)*. Nice, France: ACM Press.

McBride, Conor. (2000). Elimination with a motive. *Pages 197–216 of: International workshop on types for proofs and programs (types 2000)*.

McBride, Conor. (2010). *Ornamental algebras, algebraic ornaments*. Tech. rept. University of Strathclyde.

McKinna, James. (2006). Why dependent types matter. *Page 1 of: Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on principles of programming languages (popl 2006)*. Charleston, South Carolina, USA: ACM Press.

Mishra-Linger, Nathan, & Sheard, Tim. (2008). Erasure and polymorphism in pure type systems. *Pages 350–364 of: 11th international conference on foundations of software science and computational structures (fossacs)*. Lecture Notes in Computer Science, vol. 4962. Springer-Verlag.

Osera, Peter-Michael, Sjöberg, Vilhelm, & Zdancewic, Steve. (2012). Dependent interoperability. *Pages 3–14 of: Proceedings of the 6th workshop on programming languages meets program verification (plpv 2012)*. ACM Press.

Ou, Xinming, Tan, Gang, Mandelbaum, Yitzhak, & Walker, David. (2004). Dynamic typing with dependent types. *Pages 437–450 of: Proceedings of the ifip international conference on theoretical computer science*.

Rondon, Patrick Maxim, Kawaguchi, Ming, & Jhala, Ranjit. (2008). Liquid types. *Pages 159–169 of:* Gupta, Rajiv, & Amarasinghe, Saman P. (eds), *Proceedings of the ACM SIGPLAN conference on programming language design and implementation (pldi 2008)*. ACM Press.

Sekiyama, Taro, Nishida, Yuki, & Igarashi, Atsushi. (2015). Manifest contracts for datatypes. *Pages 195–207 of: Proceedings of the 42nd ACM SIGPLAN-SIGACT symposium on principles of programming languages (popl 2015)*. Mumbai, India: ACM Press.

Siek, Jeremy, & Taha, Walid. (2006). Gradual typing for functional languages. *Pages 81–92 of: Proceedings of the scheme and functional programming workshop*.

Siek, Jeremy G., Vitousek, Michael M., Cimini, Matteo, & Boyland, John Tang. (2015). Refined criteria for gradual typing. *Pages 274–293 of: 1st summit on advances in programming languages (snapl 2015)*.

Sozeau, Matthieu. (2010). Equations: A dependent pattern-matching compiler. *Pages 419–434 of:* Kaufmann, Matt, & Paulson, Lawrence C. (eds), *Proceedings of the 1st international conference on interactive theorem proving (itp 2010)*. Lecture Notes in Computer Science, vol. 6172. Springer-Verlag.

Sozeau, Matthieu, & Oury, Nicolas. (2008). First-class type classes. *Pages 278–293 of: Proceedings of the 21st international conference on theorem proving in higher-order logics*.

Swierstra, Wouter, & Alpuim, Joao. (2016). From proposition to program - embedding the refinement calculus in Coq. *Pages 29–44 of: Proceedings of the 13th international symposium on functional and logic programming (flops 2016)*.

Tanter, Éric, & Tabareau, Nicolas. (2015). Gradual certified programming in Coq. *Pages 26–40 of: Proceedings of the 11th ACM dynamic languages symposium (dls 2015)*. Pittsburgh, PA, USA: ACM Press.

The Coq Development Team. (2016). *The Coq proof assistant reference manual*. Version 8.6.

Thiemann, Peter, & Fennell, Luminous. (2014). Gradual typing for annotated type systems. *Pages 47–66 of:* Shao, Zhong (ed), *Proceedings of the 23rd european symposium on programming languages and systems (esop 2014)*. Lecture Notes in Computer Science, vol. 8410. Grenoble, France: Springer-Verlag.

Toro, Matías, & Tanter, Éric. (2015). Customizable gradual polymorphic effects for Scala. *Pages 935–953 of: Proceedings of the 30th ACM SIGPLAN conference on object-oriented programming systems, languages and applications (oopsla 2015)*. Pittsburgh, PA, USA: ACM Press.

Univalent Foundations Program, The. (2013). *Homotopy type theory: Univalent foundations of mathematics*. Institute for Advanced Study: `http://homotopytypetheory.org/book`.

Wadler, Philip, & Blott, Stephen. (1989). How to make ad-hoc polymorphism less ad hoc. *Pages 60–76 of: Proceedings of the 16th ACM symposium on principles of programming languages (popl 89)*. Austin, TX, USA: ACM Press.

Williams, Thomas, Dagand, Pierre-Évariste, & Rémy, Didier. (2014). Ornaments in practice. *Pages 15–24 of:* Magalhães, José Pedro, & Rompf, Tiark (eds), *Proceedings of the 10th ACM SIGPLAN workshop on generic programming (wgp 2014)*. Gothenburg, Sweden: ACM Press.

Xi, Hongwei, & Pfenning, Frank. (1998). Eliminating array bound checking through dependent types. *Pages 249–257 of: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (pldi '98)*. ACM Press.

Zimmermann, Theo, & Herbelin, Hugo. (2015). *Automatic and transparent transfer of theorems along isomorphisms in the Coq proof assistant*. arXiv:1505.05028v4.