Partial Type Equivalences for Verified Dependent Interoperability

Pierre-Evariste Dagand, Nicolas Tabareau, Éric Tanter

Partial Type Equivalences for Verified Dependent Interoperability *

Pierre-Evariste Dagand
UPMC - LIP6
Paris, France
pierre-evariste.dagand@lip6.fr

Nicolas Tabareau
Inria
Nantes, France
nicolas.tabareau@inria.fr

Éric Tanter
PLEIAD Lab
Computer Science Dept (DCC)
University of Chile - Santiago, Chile
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 paper, 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 works on homotopy type theory. Specifically, we develop the notion of *partial type equivalences* as a key foundation for dependent interoperability. 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 equivalences 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.

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 sound interaction between both type disciplines relies on a *marshalling* 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]

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

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

Following Osera et al. (2012), we illustrate the typical use cases of dependent interoperability using the simple example of simply-typed lists and dependently-typed vectors. For conciseness, we fix the element type of lists and vectors and use the type synonyms $\texttt{List}_{\mathbb{N}}$ and $\texttt{Vec}_{\mathbb{N}}\ \texttt{n}$, 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 $\texttt{max} : \texttt{List}_{\mathbb{N}} \to \mathbb{N}$ that returns the maximum element in a list. To reuse this existing function on vectors requires lifting the `max` function to the type $\forall\, \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ \texttt{n} \to \mathbb{N}$. Note that this scenario only requires losing information about the vector used as argument, so no dynamic check is needed, only a marshalling to reconstruct the corresponding list value. If the simply-typed function returns a list, *e.g.* $\texttt{rev} : \texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$, 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 $\forall\, \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ \texttt{n} \to \texttt{Vec}_{\mathbb{N}}\ \texttt{n}$ could be reused at type $\texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$. 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 $\texttt{tail} : \forall\, \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ (\texttt{n}+1) \to \texttt{Vec}_{\mathbb{N}}\ \texttt{n}$, should trigger an 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 by giving them 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 $\texttt{tail} : \texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$ should behave as a function of type $\forall\, \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ (\texttt{n}+1) \to \texttt{Vec}_{\mathbb{N}}\ \texttt{n}$ by first lifting it to this rich type, and then recasting it back to a simply-typed function `tail'` of type $\texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$. While both `tail` and `tail'` have the same type and "internal" definition, `tail'` will raise an error if called with an empty list; additionally, if the argument list is not empty, `tail'` 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 and Felleisen 2002).

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.

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 → Set :=
| 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): Set :=
 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.
```

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 OCaml, 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 sound 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 5.

Contributions In this paper, 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 *(partial) type equivalences* between programs (Section 2). Doing so, we build a conceptual as well as practical framework for relating indexed and simple types;
- By carefully segregating the computational and logical content of indexed types, we introduce a notion of *canonical equivalence* (Section 3) that identifies first-order transformations from indexed to 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 4). Using the type class mechanism of Coq, we provide a generic library for establishing partial type equivalences of dependently-typed programs;
- Finally, we illustrate our methodology through a concrete application: extracting an interoperable, certified interpreter (Section 5). Aside from exercising our library, this example is also a performance in homotopic theorem proving.

This paper 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 safety against runtime checks.

2. Partial Type Equivalences

Intuitively, dependent interoperability is about exploiting a kind of equivalence between simple and indexed types. This section formally captures such an equivalence relation, which we call *partial* because, as illustrated previously, some runtime errors might occur when crossing boundaries.

We use Coq as both a formalization vehicle and an implementation platform. We make extensive use of *type classes* (Wadler and Blott 1989) in order to define abstract structures and their properties, as well as relations among types. For instance, a partial type equivalence is a type class, whose instances must be declared in order to state an equivalence between specific types, such as $\texttt{Vec}_{\mathbb{N}}\ \texttt{n}$ and $\texttt{List}_{\mathbb{N}}$. As opposed to Haskell, type classes in Coq (Sozeau and Oury 2008) can express arbitrary properties that need to be proven

[2] In Coq terminology, a subset type is a type refined by a proposition—this is also known in the literature as refinement type (Rondon et al. 2008).

[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.

when declaring instances—for instance, the monad type class in Coq *defines and imposes* the monad laws on each instance.

In this section we progressively define the notion of partial type equivalences in a general manner. We apply partial type equivalences to the dependent interoperability context in Section 3, focusing on first-order equivalences between types of data structures. Later, in Section 4, we build higher-order partial type equivalences to interoperate between functions that manipulate such structures.

2.1 Type Equivalence

The notion of type equivalence offers a conceptual framework in which to reason about the relationships between types. Following intuitions coming from homotopy theory, 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 paper, we exploit type equivalence as a means to (constructively) state that two types "are the same". In Coq, this amounts to the following type class definition:[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, e_retr (f x) = ap f (e_sect x)
}.
```

The properties `e_sect` and `e_retr` capture the fact that `e_inv` is the inverse of `f`. The definitions use the identity function `id`, and point-wise equality between functions `==`. 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 (where `ap f` is the functorial action of `f`, transporting an equality between `x` and `y` to an equality between `f x` and `f y`).

2.2 Towards Partial Type Equivalence: The Cast Monad

As illustrated in Section 1, lifting values from simple to indexed types can fail at runtime. Thus, the type equivalences 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.

We thus define a specific `Cast` monad, which is essentially the `option` monad.[5] We use the harpoon notation `⇀` to denote a (partial) function in the `Cast` monad: `Notation "A ⇀ B" := (A → Cast B)`. The `Cast` monad is characterized by its identity `creturn` and binder `cbind`. We use the traditional do-notation. For instance, function composition in the corresponding Kleisli category, denoted $\circ_K$, is defined as follows:[6]

```
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 thus closely model the denotational objects we are interested in (here, partial functions). Crucially, the nature of these objects is reflected at the type-level: types play a guiding role in Section 2.3 below, where we lift the notion of type equivalence to the partial setting.

2.3 Partial Type Equivalence

In this section, we aim at reconciling the general notion of type equivalence with the potential for errors, as modeled by the `Cast` monad. To do so, we observe that the `Cast` monad induces a preorder. This naturally leads us to generalize the equivalence relation to operate on preorders.

Cast as a preorder with a least element. The notion of preorder with a least element is naturally defined in Coq with the following type class:

```
Class PreOrder⊥ (A:Type) :=
 { rel : A → A → Prop where "x ⪯ y" := (rel x y);
  ⊥ : A;
  rel_refl : ∀ x, x ⪯ x ;
  rel_trans : ∀ x y z, x ⪯ y → y ⪯ z → x ⪯ z;
  ⊥_is_least : ∀ a, ⊥ ⪯ a
 }.
```

The `Cast` monad induces a preorder which corresponds to equality on success values, and considers `None` as the least element of the ordering. More precisely:

```
Instance PreOrderCast A : Preorder⊥ (Cast A) :=
 {| rel := fun a a' ⇒ match a with
           | Some _ ⇒ a = a'
           | None ⇒ True
           end;
  ⊥ := None |}.
```

Any preorder on the codomain of two functions gives us a way to compare these functions pointwise:

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

Monotone functions. We must now generalize type equivalence from types equipped with an equality to types equipped with a preorder. To witness such a partial type equivalence, we shall ask for a *monotonic* function, *i.e.* a function that preserves the preorder relation (and the least element).[7]

```
Record monotone_function X Y `{Preorder⊥ X}
 `{Preorder⊥ Y} := Build_Mon
  { f_ord :> X → Y ;
   mon :∀ x y, x ⪯ y → f_ord x ⪯ f_ord y;
   mon_p : f_ord ⊥ ⪯ ⊥
  }.
Notation "X ⟶ Y" := (monotone_function X Y).
```

Monotonicity is expressed through the functorial action `f.(mon)`, thus following and generalizing the functorial action `ap f` of (total) type equivalences (Section 2.1). 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 `X ⟶ Y` to `X → Y`: we can transparently manipulate monotone functions as standard functions.

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

[5] We discuss some specificities of the `Cast` monad in Section 5.5.

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

[7] In Coq, back-quoted parameters are nameless. Records differ from type classes in that they are not involved in (implicit) instance resolution; other than that, type classes are essentially records (Sozeau and Oury 2008).

Partial equivalence. We now capture the notion of a partial equivalence between two preorders `A` and `B`. The definition of `IsPartialEquiv` is directly derived from its total counterpart, by replacing functions with monotone functions, and equality by the preorder relation $\preceq$.

```
Class IsPartialEquiv (A B : Type) (f:A ⟶ B)
`{Preorder⊥ A} `{Preorder⊥ B} := {
 pe_inv : B ⟶ A ;
 pe_sect : pe_inv ∘ f ⪯ id;
 pe_retr : f ∘ pe_inv ⪯ id;
 pe_adj  : ∀ x, pe_retr (f x) = ap f (pe_sect x)
}.
```

2.4 Partial Type Equivalence in the Kleisli Category

The preorder over `Cast` types yields the expected notion of type equivalence in the Kleisli (bi-)category defined in Section 2.2. Composition amounts to monadic composition $\circ_K$, and identity to monadic identity `creturn`. We substantiate this intuition by specifying a monadic equivalence to be:

```
Class IsPartialEquivK (A B : Type) (f:A ⇀ B) := {
 pek_inv : B ⇀ A ;
 pek_sect : pek_inv ∘K f ⪯ creturn ;
 pek_retr : f ∘K pek_inv ⪯ creturn;
 pek_adj : ∀ x,
    ((pek_sect ∘V (id2 f)) ∘H idL f) x =
    (α f pek_inv f ∘H ((id2 f) ∘V pek_retr) ∘H idR f) x
}.
```

Note that the definition of `pek_adj` is more complicated than for partial equivalences, as explained in Section 2.5 below.

As a sanity check, we can prove that lifting an equivalence yields a partial type equivalence in the Kleisli category, meaning in particular that `pek_adj` is a conservative extension of `e_adj` to the Kleisli category.

```
Definition EquivToPartialEquivK A B (f :A → B) :
 IsEquiv f → IsPartialEquivK (clift f).
```

2.5 A (Bi-)Categorical Detour

The various notions of equivalence presented above (total, partial, and partial in the Kleisli category of the `Cast` monad) follow a common pattern—they are all instances of the concept of adjunction in a bicategory, coming from the seminal work of Bénabou (1967). Recall that 2-categories generalize categories by introducing a notion of 2-cells, *i.e.* morphisms between morphisms, but letting compatibility laws hold strictly. In the setting of type theory, the fact that compatibility laws are strict means that they hold definitionally by conversion in the system. Bicategories generalize 2-categories by allowing compatibility laws not to be defined strictly, but up-to an invertible 2-cell.

Relaxing compatibility laws up-to invertible 2-cells is not necessary to describe (total) type equivalence because associativity and identity laws hold strictly on functions between types, as they are directly captured by β-reduction.

For partial type equivalence, strict laws for composition also hold because we are dealing with proofs of monotonicity that are irrelevant in the sense that they are stated on a notion of preorder that lives in `Prop`. Note that we could have defined a proof-relevant preorder and monotonicity condition, in which case the need to go to a bicategorical setting would have manifested itself at this stage.

When considering the Kleisli category induced by the `Cast` monad, however, it is not possible to avoid bicategories anymore because, for instance, associativity of Kleisli composition does not hold strictly. Indeed, the different order in which arguments are evaluated (*i.e.* in which effects are performed) matters, and so associativity only holds up to a proof term. That is, there is a term to make explicit the associativity of composition (we express it from left-to-right but the converse also holds):

```
Definition α {X Y Z T: Type} (f:X ⇀ Y) (g:Y ⇀ Z) (h:Z ⇀ T):
  h ∘K (g ∘K f)) ⪯ (h ∘K g) ∘K f.
```

In the same way, we need to exhibit the usual morphisms:

```
idR f : creturn ∘K f ⪯ f                          (right identity law)
idL f : f ∘K creturn ⪯ f                          (left identity law)
∘V : f ⪯ f' → g ⪯ g'
       → g ∘K f ⪯ g' ∘K f'                        (vertical composition)
∘H : e ⪯ f → f ⪯ g → e ⪯ g                        (horizontal composition)
id2 f : f ⪯ f                                     (2-identities)
```

It follows that the definition of `pek_adj` is merely the expected formulation of the compatibility of an adjunction in a bicategory.

3. Partial Type Equivalence for Dependent Interoperability

We now exploit partial type equivalences to setup a verified framework for dependent interoperability. In this context, we are specifically interested in partial equivalences between indexed types, such as $\texttt{Vec}_{\mathbb{N}}$, and simple types, such as $\texttt{List}_{\mathbb{N}}$. We call this kind of partial type equivalence a *dependent equivalence*.

A major insight of this work is that a dependent equivalence can be defined by composition of two different kinds of type equivalences, using subset types as intermediaries:

- a *total* equivalence between indexed types and subset types of the form {c:C & P c} whose logical content P—*i.e.* static invariants—is carefully quarantined from their computational content C—*i.e.* runtime representation;
- a *partial* equivalence between subset types and simple types, in the Kleisli category of the `Cast` monad.

The resulting dependent equivalence is therefore also a partial type equivalence in the Kleisli category.

For instance, to establish the equivalence between $\texttt{Vec}_{\mathbb{N}}$ and $\texttt{List}_{\mathbb{N}}$, we exploit the subset type $\{\texttt{l:List}_{\mathbb{N}}\ \&\ \texttt{length l = n}\}$, which captures the *meaning* of the index of $\texttt{Vec}_{\mathbb{N}}$.

3.1 Partial Equivalence Relations

The equivalence classes defined in Section 2 characterize 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 $\texttt{A} \simeq \texttt{B}$. The record thus encapsulates the equivalence function `e_fun` and defines a type relation:

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

For partial type equivalences in the Kleisli category of the `Cast` monad, we similarly define the record type and notation:

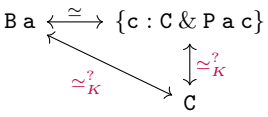
```
Record PartialEquivK (A B : Type) := {
 pek_fun : A ⇀ B;
 pek_isequiv: IsPartialEquivK pek_fun
}
Notation "A ≃?K B" := (PartialEquivK A B).
```

For dependent interoperability, however, we want to consider *canonical* dependent equivalences between index and simple types, so as to automate the lifting between both worlds. This is particularly important to perform complex, automatic lifting of functions, as described in Section 4. For instance, lifting a function that operate on $\texttt{Vec}_{\mathbb{N}}$ to an equivalent function that operates on $\texttt{List}_{\mathbb{N}}$ implies "looking up" the canonical equivalence between these types.

Technically, this means that canonical partial equivalences need to be presented to Coq as a type class, in order to take advantage of the automatic instance resolution mechanism they offer:

```
Class CanonicalPartialEquiv (A B : Type) := {
 pe_fun : A ⟶ B;
 pe_isequiv: IsPartialEquiv pe_fun
}
Notation "A ≃? B" := (CanonicalPartialEquiv A B).
```

The strategy to establish dependent equivalences via subset types can be depicted as follows, for a given index a:



The following subsections describe the two (orthogonal) equivalences below, before defining their (diagonal) composition.

In the diagram above, and the rest of this paper, 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`.

Remark: HProp/HSet vs. Prop/Set The reader might expect `P` to end in `Prop`, not `Type`. 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 however not correct. On the one hand, it is too restrictive because some properties are proof irrelevant but cannot be typed in `Prop` (Mishra-Linger and Sheard 2008). 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` is irrelevant is independent from the theory of Coq as it corresponds to the Uniqueness of Identity Proofs (UIP) axiom.

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 X := {isHSet :> ∀ (a b : X), IsHProp (a = b)}.
```

In the rest of the code, we (abusively) write `T : HProp` for a type `T : Type` for which there exists an instance of `IsHProp T` (and similarly for `HSet`). Therefore, the logical proposition capturing the meaning of the index is hereafter written `P : A → C → HProp`. Similarly, since simple types represent data structures, which are sets, we write `C : HSet`.

3.2 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 equivalence 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`:

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

Recall from Section 2.1 that establishing this total equivalence requires providing two functions:

```
vector_to_list n: Vec_ℕ n → { l : List_ℕ & length l = n }
list_to_vector n: { l : List_ℕ & length l = n } → Vec_ℕ n
```

These functions capture the computational content of the conversion between the two data structures.[8] 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. We come back to the question of proving the coherence in Section 3.5, and we discuss some useful proof techniques for type conversions in Section 5.6.

3.3 Equivalence Between Subset and Simple Types

The second equivalence we exploit is a *partial* type equivalence in the Kleisli category of the `Cast` monad between subset types and simple types such as, for a given `n`:

$$\{\ \texttt{l} : \texttt{List}_{\mathbb{N}}\ \&\ \texttt{length l = n}\ \} \simeq^?_K \texttt{List}_{\mathbb{N}}$$

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.

Recently, Tanter and Tabareau (2015) developed an approach in Coq to cast a value of any simple type `C` to a value of a subset type `{c:C & P c}` for any decidable predicate `P`. Using decidability, the authors perform the type cast through a runtime check, relying on an axiom to capture the potential for cast failure. In the monadic setting we adopt here, there is no need for axioms anymore, and their technique amounts to establishing a partial type equivalence $\{\texttt{c:C \& P c}\} \simeq^?_K \texttt{C}$. We capture this partial type equivalence between subset types and simple types with the following instance:

```
Definition Checkable_PEquivK (C : HSet) (P : C → HProp)
`{∀ c, Checkable (P c)} : {c:C & P c} ≃?K C :=
 {| pek_fun := (clift π1 : {c:C & P c} ⇀ C);
    pek_isequiv := {| pek_inv := to_subset |}|}.
```

Instead of imposing actual *decidability*, the `Checkable` type class (defined in Appendix A) 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`).

The equivalence function `pek_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 cast operator of Tanter and Tabareau (2015):

```
Definition to_subset {C : HSet} {P : C → HProp}
`{∀ c, Checkable (P c)}: C ⇀ ({c:C & P c}) :=
  fun c ⇒
   match dec checkP with
   | inl p ⇒ Some (c; convert p)
   | inr _ ⇒ None
   end.
```

[8] 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.

`to_subset` applies the sound approximation decision procedure for the embedded logical proposition. If it succeeds, the proof of success of the approximation is converted (by implication) to a proof of the property. Otherwise an error is raised.

Note that proof irrelevance of `P c` is crucial, because when going from the subset type `{c:C & P c}` to the simple type `C` and back, the element of `P c` is inferred by the approximation decision, 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 equivalence is established.

3.4 Equivalence Between Dependent and Simple Types

We now define a partial equivalence between dependent and simple types by composing the two equivalences described above. The *dependent equivalence* class below captures the necessary requirements for two structures to interoperate. `DepEquiv` corresponds to a *first order* dependent interoperability, in as much as it only relates data structures. In Section 4, we shall develop higher-order dependent equivalences, 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 partial equivalences automatically.

```
Class DepEquiv (A : Type) (B: A → Type) (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 pek_fun) (e_fun b) = Some a;
}.
Notation "B ≈ C" := (DepEquiv _ B C).
```

(The index type `A` can always be inferred from the context so the notation ≈ omits it.) A key ingredient to establishing a dependent equivalence between the type family `B` and the simple type `C` is the property `P` that connects the two equivalences. Note that the partial and total equivalences with the subset type are lifted to point-wise equivalences, *i.e.* they must hold for all indices `a`.[9]

The `DepEquiv` class also includes an index synthesis function, f_{ca}, which recovers a canonical index from a data of simple type. In the case of an $\texttt{List}_{\mathbb{N}}$, it is always possible to compute its length, but as we will see in the case of stack machine instructions (Section 5), synthesizing an index may fail. The f_{ca} function is used for defining higher-order equivalences, *i.e.* for automatically lifting functions (Section 4). The property $P_{f_{ca}}$ states that if we have a value `c : C` that was obtained through the equivalence from a value of type `B a`, then f_{ca} is defined on `c` and recovers the original index `a`.

Finally, for all index `a`, `B ≈ C` is a partial type equivalence in the Kleisli category of the `Cast` monad:

```
Definition DepEquiv_PEK (A : Type) (B : A → Type)
    (C : HSet) `{B ≈ C} (a:A) : B a ≃?K C :=
{| pek_fun := to_simpl;
   pek_isequiv := {| pek_inv := to_dep a |} |}.
```

The functions used to establish the partial equivalence are `to_simpl`, which is the standard composition of the two equivalence functions `pek_fun ∘ e_fun`, and the function `to_dep`, which is the Kleisli composition of the inverse functions, `(clift e_inv) ∘K pek_inv`.

3.5 Simplifying the Definition of a Dependent Equivalence

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 `IsDepEquiv` that exploits this and creates a dependent equivalence without requiring the extra coherences `e_adj` or `pek_adj`.

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

```
Definition IsDepEquiv {A: Type} (B: A → Type) (C:HSet)
 (P: A → C → HProp) `{∀ a c, Checkable (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 _ b).1 = Some a) → B ≈ C.
```

Using `IsDepEquiv`, establishing a new dependent interoperability between two types such as $\texttt{Vec}_{\mathbb{N}}$ and $\texttt{List}_{\mathbb{N}}$ boils down to providing a checkable predicate, two inverse conversion functions (as in Section 3.5), and the index synthesis function (`length`). The programmer must then prove three equations corresponding to the properties of conversions and that of the index synthesis function.

Frequently, the checkable predicate merely states that the synthesized index is equal to the proposed index (*i.e.* `P := fun a c ⇒ f_ca_eq c = Some a`). We provide another convenient instance constructor `DepEquiv_eq`, specialized to handle this situation. Declaring the canonical dependent equivalence between $\texttt{Vec}_{\mathbb{N}}$ and $\texttt{List}_{\mathbb{N}}$ amounts to:

```
Instance DepEquiv_vector_list : Vec_ℕ ≈ List_ℕ :=
  DepEquiv_eq Vec_ℕ List_ℕ (clift length)
              vector_to_list list_to_vector.
```

4. Higher-Order Partial Type Equivalence

Having defined first-order dependent equivalences, which relate indexed types (*e.g.* $\texttt{Vec}_{\mathbb{N}}$) and simple types (*e.g.* $\texttt{List}_{\mathbb{N}}$), we now turn to *higher-order* dependent equivalences, which rely on higher-order partial type equivalences. These higher-order equivalences relate partial functions over simple types, such as $\texttt{List}_{\mathbb{N}} \rightharpoonup \texttt{List}_{\mathbb{N}}$, to partially-equivalent functions over indexed types, such as $\forall\ \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ (\texttt{n} + 1) \rightharpoonup \texttt{Vec}_{\mathbb{N}}\ \texttt{n}$.

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

4.1 Defining A Higher-Order Dependent Interoperability

Consider that two first-order dependent equivalences $\texttt{B}_1 \approx \texttt{C}_1$ and $\texttt{B}_2 \approx \texttt{C}_2$ have been previously established. We can construct a higher-order partial type equivalence between functions of type $\forall\ \texttt{a:A}, \texttt{B}_1\ \texttt{a} \rightharpoonup \texttt{B}_2\ \texttt{a}$ and functions of type $\texttt{C}_1 \rightharpoonup \texttt{C}_2$:

```
Instance HODepEquiv {A: Type}
 {B₁: A → Type} {C₁: HSet} `{B₁ ≈ C₁}
 {B₂: A → Type} {C₂: HSet} `{B₂ ≈ C₂} :
 (∀ a:A, B₁ a ⇀ B₂ a) ≃? (C₁ ⇀ C₂) :=

{| pe_fun := fun f ⇒ to_simpl_dom
                      (fun a b ⇒ x ← f a b;
                                  to_simpl x)) _ ;
  pe_isequiv := {| pe_inv :=
```

[9] To define a dependent equivalence, Coq must also be able to infer that the type `C` is an `HSet`. In practice, it is convenient to exploit Hedberg's theorem (Univalent Foundations Program 2013, Section 7.2), which states that decidable equality on `T` (which is easier to prove) implies `isHSet T`.

```
                fun f a b ⇒ x ← to_dep_dom f a b;
                             to_dep _ x) _ |}|}.
```

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

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

```
Definition to_dep_dom {A X} {B: A → Type} {C : HSet}
    `{B ≈ C} (f: C ⇀ X) (a:A) : B a ⇀ X := 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 `DepEquiv` type class.

```
Definition to_simpl_dom {A X} {B: A → Type} {C : HSet}
    `{B ≈ C} (f : ∀ a:A, B a ⇀ X) : C ⇀ X :=
 fun c ⇒ a ← f_ca c;
         b ← to_dep a c ;
         f a b.
```

Crucially, the proof that `HODepEquiv` is a partial equivalence 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 5.6.

As the coherence condition of `HODepEquiv` 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 can not be proven in Coq for the moment, because equality is defined as an inductive type, and the dependent product is of a coinductive nature.

In order to cope with pure functions of type `∀ a, B a → C a`, we first embed the pure function into the monadic setting and then apply a partial equivalence:

```
Definition lift {A} {B₁: A → Type} {B₂ : A → Type}
   {C₁ C₂ : HSet} `{∀ a, B₁ a ⇀ B₂ a ≃? C₁ ⇀ C₂} :
  (∀ a, B₁ a → B₂ a) → C₁ ⇀ C₂ :=
 fun f ⇒ pe_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:

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

is a mere `lift` away:

```
Definition map_simpl (f : ℕ → ℕ) : list ℕ ⇀ list ℕ
 := lift (Vec_ℕ.map f).
```

Note that it is however not (yet) possible to lift the tail function `Vec_ℕ.tl : ∀ n, Vec_ℕ (S n) → Vec_ℕ n` because there is no dependent equivalence between `Vec_ℕ (S n)` and `List_ℕ`. Fortunately, the framework is extensible and we will see in the next section how to deal with this example, among others.

4.2 A Library of Higher-Order Dependent Equivalences

`HODepEquiv` is but one instance of an extensible library of higher-order dependent equivalence classes. One of the benefits of our approach to dependent interoperability is the flexibility of the framework. Automation of higher-order dependent equivalences 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. `HODepEquiv` 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 `Vec_ℕ.tl : ∀ n, Vec_ℕ (S n) → Vec_ℕ n` to a function of type `List_ℕ → List_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 : Type} (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 `DepEquiv` 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 DepEquivInj (A A' : Type) (B : A → Type)
  (C : HSet) (f : A' → A) `{IsInjective f} `{B ≈ C} :
   (fun a ⇒ B (f a)) ≈ C
```

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

```
Definition pop : list ℕ ⇀ list ℕ := lift Vec_ℕ.tl.
```

As expected, when applied to the empty list, the function `pop` returns `None`, which corresponds to the error of the `Cast` monad.[10] In the other direction, we can as easily lift a `pop` function on lists to the dependent type `∀ n, Vec_ℕ (S n) ⇀ Vec_ℕ n`. This function can only be applied to a non-empty vector, but if it does not return a vector of a length reduced by one, a cast error is reported.

Composing equivalences. 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', B₁ a → B₂ a a' ⇀ B₃ a a'
```

We can define an instance of $\simeq^?$ to form a new partial equivalence on `∀ a a', B₁ a → B₂ a a' ⇀ B₃ a a'` from a partial equivalence on `∀ a', B₂ a a' ⇀ B₃ a a'`, for a fixed `a`, provided that we have established that `B₁ ≈ C₁`:

```
Instance HODepEquiv₂ A A' B₁ B₂ B₃ C₁ C₂ C₃
  `{∀ a, ((∀ a':A', B₂ a a' ⇀ B₃ a a') ≃? (C₂ ⇀ C₃))}
  `{B₁ ≈ C₁}:
 (∀ a a', B₁ a → B₂ a a' ⇀ B₃ a a') ≃? (C₁ → C₂ ⇀ C₃).
```

For space reasons, 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 explained in Section 5, the example involves composing two dependent equivalences, one on instructions and one on stacks.

Index dependencies. It is sometimes necessary to reorder arguments in order to be able to compose equivalences, 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 partial equivalence:

[10] We come back to an improvement of the error message in Section 5.5

```
Instance HODepEquiv₂_sym A A' B₁ B₂ B₃ C₁ C₂ C₃
'{∀ 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 must be used as a last resort, or one would introduce cycles during type class resolution. The stack machine example in Section 5 also exploits this instance.

5. 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: ∀ 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 equivalences, one dealing with instructions and the other dealing with stacks. Once those equivalences 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. Thus, the automation of higher-order lifting needs to be able to linearize such indexing flow and turning them into sequential checks.

In this process, users are only asked to provide the two first-order type equivalences specific to their target domain, `dstack` ≈ `List_ℕ` and ∀ `n`, `dinstr n` ≈ `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 equivalences (Section 4).

5.1 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 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, Section 3.2). However, using specific domain knowledge, the programmer can craft an equivalence with a list, along the lines of the equivalence 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.[11] Crucially, this transformation involves a change of representation: we move from tuples to lists. For our framework to automatically handle this transformation, we declare the suitable instance of dependent equivalence:

```
Instance DepEquiv_dstack : dstack ≈ List_ℕ :=
    DepEquiv_eq dstack List_ℕ (clift length)
                dstack_to_list list_to_dstack.
```

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

5.2 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 → Set :=
| 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) : Cast ℕ :=
 match i with
  | NConst _ ⇒ Some (S n)
  | NPlus ⇒ match n with
          | S (S n) ⇒ Some (S n)
          | _ ⇒ None
        end
 end.
```

The dependent equivalence 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 equivalence through a suitable instance declaration:

```
Instance DepEquiv_instr n : (dinstr n) ≈ instr :=
 DepEquiv_eq (dinstr n) instr (instr_index n)
             (dinstr_to_instr n) (instr_to_dinstr n).
```

5.3 Lifting the Interpreter

Having specified our domain-specific equivalences, we are left to initiate the instance resolution so as to *automatically* obtain the desired, partially-equivalent lifting of the interpreter `exec`. To do so, we simply appeal to the `lift2` operator, which is similar to `lift` from Section 4.1 save for the fact that it deals with two-index types:

[11] 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.

```
lift2 A A' B1 B2 B3 C1 C2 C3
 '{∀ a a', B1 a a' → B2 a a' ⇀ B3 a a' ≃? 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 partially-equivalent to the dependent program we started from.

5.4 Diving into the Generated Term

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

```
simple_exec = lift2
 (HODepEquiv2_sym
   (HODepEquiv2
     (fun a : ℕ ⇒ HODepEquiv (DepEquiv_instr a)
                             DepEquiv_stack)
     DepEquiv_stack)) exec
```

We witness three generic transformations of the function: `HODepEquiv2_sym`, which has reordered the input arguments so as to first determine the size of the input stack; `HODepEquiv2`, which has made the size of the input list available to subsequent transformations; and `HODepEquiv`, 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
```

5.5 Extraction to OCaml

In Section 2.2, we introduced the `Cast` monad as *essentially* the option monad. For practical purposes, the failure constructor of `Cast` takes additional arguments for producing informative error messages: we capture the type we are trying to cast 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 `Cast` 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). 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 cast errors—while preserving the software engineering benefits of not imposing a monadic framework on external components. The definition of the extraction of the `Cast` monad is provided in Appendix B.

We can now revisit the interaction with the extracted function:

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

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

5.6 A Homotopical Detour

Before concluding, we 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 3 and 4) or on domain-specific equivalences (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 2):

```
Definition transport {A : Type} (P : A → Type) {x y : A}
 (p : x = y) : P x → P y.
Notation "p # x" := (transport _ p x).
```

Transport is trivially implemented by path induction, 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.[12] 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.
```

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

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 prove auxiliary lemmas on transport 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 n0)) = (NConst n0; ap Some e).
```

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

6. 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 Matthews and Findler (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 partial type equivalences, our framework is able to exploit them automatically. In particular, we do not require a one-to-one correspondence between constructors: the equivalence is established at the type level, giving the user the freedom to implement potentially partial transformations. We also account for more general equivalences 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 type equivalences 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 (Knowles and Flanagan 2010)). This means that the authors do not address the issues related to indexed types, including that of providing correct marshalling functions between representations, which is a core challenge of dependent interoperability.

Casts 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 sound casts 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 equivalence builds upon the idea of casting 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 cast errors, we operate within a `Cast` 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 equivalences only abstractly rely on casting 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 and 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 equivalences between subset and simple types in the Kleisli category of the `Cast` monad.

Gradual typing. Multi-language semantics are directly related to *gradual typing* (Siek and 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) and security typing (Disney and Flanagan 2011; Fennell and Thiemann 2013). 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, casting requires marshalling (Osera et al. 2012).

In our framework, casting (which we termed "lifting") must be explicitly summoned. However, as already observed by Tanter and Tabareau (2015), the implicit coercions of Coq 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 equivalences, which relate types of different precision.

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 ingredients of the ornamental toolbox are algebraic ornaments (McBride 2010) and relational ornaments (Ko and Gibbons 2013). From an inductive type and an algebra (over an endofunctor on $\mathcal{S}et$ for algebraic ornaments, over an endofunctor on $\mathcal{R}el$ 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) proof of type equivalence. Conceptually, partial type equivalences subsume ornaments in the sense that an ornament automatically gives rise to a partial type equivalence. However, ornaments are restricted to inductive types, while partial type equivalences 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 and 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 that are difficult to formally rationalize.

By focusing on type equivalences, 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 (Rondon et al. 2008). Indeed, dependent equivalences 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. 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 equivalences. Besides, instead of introducing another manifest contract calculus and painstakenly developing its meta-theory, we leverage full-spectrum dependent types to internalize the cast machinery through partial type equivalences.

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), or the step-wise refinement of specifications down to correct-by-construction programs (Delaware et al. 2015; Swierstra and Alpuim 2016). Our work differs from the former 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 equivalences. 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.

7. Conclusion

In this paper, we have given a semantic account of dependent interoperability through the notion of partial type equivalence. Our definitions were set up to be directly mechanizable: this resulted in a library of generic equivalence-preserving program transformations and generic proofs of said equivalences. To our knowledge, this is the first implementation 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 type equivalence 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 type equivalences are expressed as type classes, allowing users to plug and immediately play with their domain-specific equivalences. We also expose the cast operators through this mechanism. In effect, lifting programs is implemented through a logic program, critically rely-

ing 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 equivalences (typically, `DepEquiv_eq`) are genuine *definition sites*. For instance, the input list of the stack machine *defines* its associated index. Other equivalences 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. As hinted at earlier, the second argument of the typed instruction can be computed from the first one, thus witnessing a *use-definition chain*. Conceptually, lifting a dependently-typed program consists in performing a topological sort on this dataflow graph.

Taking full advantage of this representation opens many avenues for generalizations. For instance, our current definition of dependent equivalences insists on being able to recover an index from a raw value through the mandatory $\mathtt{f}_{ca} : \mathtt{C} \rightharpoonup \mathtt{A}$. As such, this precludes the definition of many equivalences, such as the equivalence 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 marshalling 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. 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.

Coq Formalization. The full Coq formalization, with documentation, is available at `http://coqhott.github.io/DICoq/`. It has been developed using the 8.5 release of Coq (The Coq Development Team 2015).

Acknowledgments. We thank the anonymous reviewers for their *constructive* feedback.

References

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Proceedings of the ACM Workshop on Programming Languages meets Program Verification (PLPV 2007)*, pages 57–68, Freiburg, Germany, Oct. 2007.

S. Awodey and A. Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

F. Bañados, R. Garcia, and É. Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 283–295, Gothenburg, Sweden, Sept. 2014. ACM Press.

J. Bénabou. Introduction to bicategories. In *Reports of the Midwest Category Seminar*, pages 1–77. Springer-Verlag, 1967.

E. Brady, C. McBride, and J. McKinna. *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, chapter Inductive Families Need Not Store Their Indices, pages 115–129. Springer-Verlag, 2004.

A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013.

C. Cohen, M. Dénès, and A. Mörtberg. Refinements for free! In *Proceedings of the 3rd International Conference on Certified Programs and Proofs (CPP 2013)*, pages 147–162, Melbourne, Australia, Dec. 2013.

P.-E. Dagand and C. McBride. Transporting functions across ornaments. In *Proceedings of the 17th ACM SIGPLAN Conference on Functional Programming (ICFP 2012)*, pages 103–114, Copenhagen, Denmark, Sept. 2012. ACM Press.

B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 689–700, Mumbai, India, Jan. 2015. ACM Press.

T. Disney and C. Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

L. Fennell and P. Thiemann. Gradual security typing with references. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*, pages 224–239, June 2013.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002)*, pages 48–59, Pittsburgh, PA, USA, Sept. 2002. ACM Press.

R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 429–442, St Petersburg, FL, USA, Jan. 2016. ACM Press.

G. Gonthier and A. Mahbouhi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

J. M. E. Hyland. First steps in synthetic domain theory. In *Proceedings of the International Conference on Category Theory*, pages 131–156, Como, Italy, July 1991. Springer-Verlag.

K. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2):Article n.6, Jan. 2010.

H.-S. Ko and J. Gibbons. Relational algebraic ornaments. In *Proceedings of the ACM SIGPLAN Workshop on Dependently Typed Programming (DTP 2013)*, pages 37–48. ACM Press, 2013.

P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, pages 3–10, Nice, France, Jan. 2007. ACM Press.

C. McBride. Ornamental algebras, algebraic ornaments. Technical report, University of Strathclyde, 2010. URL `https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/Ornament.pdf`.

N. Mishra-Linger and T. Sheard. Erasure and polymorphism in pure type systems. In *11th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer-Verlag, 2008.

P.-M. Osera, V. Sjöberg, and S. Zdancewic. Dependent interoperability. In *Proceedings of the 6th workshop on Programming Languages Meets Program Verification (PLPV 2012)*, pages 3–14. ACM Press, 2012.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004.

P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 159–169. ACM Press, June 2008.

T. Sekiyama, Y. Nishida, and A. Igarashi. Manifest contracts for datatypes. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages (POPL 2015), pages 195–207, Mumbai, India, Jan. 2015. ACM Press.

J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

M. Sozeau and N. Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher-Order Logics*, pages 278–293, Montreal, Canada, Aug. 2008.

W. Swierstra and J. Alpuim. From proposition to program - embedding the refinement calculus in Coq. In *Proceedings of the 13th International Symposium on Functional and Logic Programming (FLOPS 2016)*, pages 29–44, Kochi, Japan, Mar. 2016.

É. Tanter and N. Tabareau. Gradual certified programming in Coq. In *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*, pages 26–40, Pittsburgh, PA, USA, Oct. 2015. ACM Press.

The Coq Development Team. *The Coq proof assistant reference manual*. 2015. URL `http://coq.inria.fr`. Version 8.5.

T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 89)*, pages 60–76, Austin, TX, USA, Jan. 1989. ACM Press.

T. Williams, P. Dagand, and D. Rémy. Ornaments in practice. In J. P. Magalhães and T. Rompf, editors, *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP 2014)*, pages 15–24, Gothenburg, Sweden, Aug. 2014. ACM Press.

A. Decidable and Checkable

A proposition `A` is an instance of the `Decidable` type class when there exists a function that return either a proof of `A` or a proof of `not A`. We restrict the use of the `Decidable` type class to `HProp` to force the element computed by the decidability function to be irrelevant.

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

A proposition is `Checkable` when there exists a decidable proposition `checkP` that implies it.

```
Class Checkable (A : HProp) := {
  checkP : HProp;
  checkP_dec : Decidable checkP ;
  convert : checkP → A;
  is_hP_c :> IsHProp A }.
```

B. Cast Monad

The `Cast` Monad is a refinement of the Maybe monad, that allows to collect information on the error.

```
Inductive _Cast A info :=
 | Some : A → _Cast A info
 | Fail : info → _Cast A info.
```

Here, we want to collect an error message in the form of a `string`. However, we need this extra piece of information to be irrelevant. For that, we use the *propositional truncation* `Trunc`, as introduced by Awodey and Bauer (Awodey and Bauer 2004)—but in the form defined in the HoTT book (Univalent Foundations Program 2013). This allows us to state formally that the error message is irrelevant while preserving consistency and compatibility with univalence.

```
Definition Cast A := _Cast A (Trunc string).
```

We have standard monadic functions and notations:

```
Definition clift A B : (A → B) → A → Cast B :=
 fun f a ⇒ Some (f a).

Definition cbind A B : (A → Cast B) → Cast A → Cast B :=
fun f a ⇒
 match a with
   Some a ⇒ f a
  | Fail _ s t ⇒ Fail _ s t
 end.

Notation "x ← e1; e2" := cbind (fun x ⇒ e2) e1.
```

We use extraction to provide a direct style extraction of the `Cast` monad in OCaml, using runtime exceptions. The success constructor of the monad is simply erased, and its failure constructor is projected to a runtime exception where argument of the failure constructor are used as the error message.

```
(* Transparent extraction of Cast:
   - if Some t, then extract plain t
   - if Fail, then fail with a runtime cast exception *)

Extract Inductive Cast ⇒
   "" [ "" "(let f s = failwith
      (String.concat """" ([""Cast failure: ""]@
      (List.map (String.make 1) s))) in  f)"]
   "(let new_pattern some none = some in
         new_pattern)".
```