Extended Abstract @ ML Family Workshop 2015

Lost in Extraction, Recovered

An Approach to Compensate for the Loss of Properties and Type Dependencies when Extracting Coq Components to OCaml

Éric Tanter
PLEIAD Lab
Computer Science Department (DCC)
University of Chile, Santiago, Chile
etanter@dcc.uchile.cl

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

Abstract

Extracting Coq programs to OCaml enables a model of software development in which some critical components of a system can be developed in Coq and proven correct, before being extracted to OCaml and integrated with the rest of the system. However, due to the gap between Coq and OCaml, key features like properties and type dependencies disappear upon extraction, leading to potentially unsafe and unsound executions of certified components. We describe an approach to protect extracted components through mostly automatic generation of runtime checks, ensuring that the assumptions made by certified components are checked dynamically when interacting with non-certified components.

1. Introduction

OCaml Programmers should be able to exploit program extraction to integrate Coq certified components within their systems, while ensuring that the assumptions made by the certified components are enforced. However, extraction of Coq components to OCaml suffers from a number of limitations that can lead to unsound or even unsafe executions, *e.g.* yielding segmentation faults.

While some unsafe executions can be produced by using effectful functions as arguments to functions extracted from Coq—because referential transparency is broken—we focus on two fundamental issues that manifest even when the OCaml code is pure: properties and type dependencies.

Let us consider *subset types*, the canonical way to attach a property to a value [1, 5]. Subset types are of the form $\{a{:}A \mid P\ a\}$, denoting the elements a of type A for which property $P\ a$ holds. More precisely, an inhabitant of $\{a{:}A \mid P\ a\}$ is a dependent pair $(a\ ;\ p)$, where a is a term of type A, and p is a proof term of type $P\ a$.

Suppose a function divide of type $\mathsf{nat} \to \{n{:}\mathsf{nat} \mid n > 0\} \to \mathsf{nat}$ defined in Coq. To define divide, the programmer works under the assumption that the second argument is strictly positive. However, this guarantee is lost when extracting the function: Coq establishes a difference between programs (in `Type`), which have computational content, and proofs (in `Prop`), which are devoid of computational meaning and are therefore erased during extraction. Ideally, we would like to extract divide to an OCaml function that checks that the second argument is indeed positive, or fails otherwise:

```
let divide m n = if n > 0 then ...
                 else failwith "invalid argument"
```

This would allow the original code (`...`) to be executed in a sound context, where `n > 0` indeed holds. Such an approach to transfer properties as runtime checks can be supported as long as the stated property is decidable.

The situation can be even worse with type dependencies, since extracting dependent structures to OCaml typically introduces unsafe operations. Consequently, it is easy for an OCaml client to produce segfaults. Consider the following example adapted from CPDT [1], in which the types of the instructions for a stack machine are explicit about their effect on the size of the stack:

```
Inductive dinstr : nat → nat → Set :=
| IConst : ∀ n, nat → 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 nats. 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 : nat) : Set :=
  match n with
    | O ⇒ unit
    | S n' ⇒ nat × dstack n'
  end%type.
```

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 relies on unsafe coercions:

```
(* exec : int -> int -> dinstr -> dstack -> dstack *)
let exec n m i s =
  match i with
  | IConst (n0, n1) -> Obj.magic (Pair (n1, s))
  | IPlus n0 ->
    let Pair (arg1, t) = Obj.magic s in
    let Pair (arg2, s') = t in
      Obj.magic (Pair ((add arg1 arg2), s'))
```

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 `'a -> '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:[1]

```
# exec 0 0 (IPlus 0) [];;
Segmentation fault: 11
```

While it might be possible to preserve some type dependencies by exploiting advanced features of the target language type system (in particular, GADTs in OCaml), there will inevitably be cases where the dependencies expressed in Coq outsmart the target type system and extraction relies on unsafe features.

We propose an approach in which a richly-typed Coq function can be *casted*[2] to a function with a standard type signature that matches the target language type system, in a way that automatically embeds the necessary runtime checks in the casted function.

2. Recovering Properties

We develop safe casts[3] for Coq, paving the way for gradual certified programming; notably, we show that this is feasible entirely within standard Coq, without extending the underlying theory and implementation. Note that the focus of this presentation is on OCaml extraction, not on gradual certified programming in Coq (the latter is presented in full details in [6]).

Concretely, we can cast divide from type $\mathsf{nat} \to \{n{:}\mathsf{nat} \mid n > 0\} \to \mathsf{nat}$ to a function divide' with the plain type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ in a safe manner, using the $?^d$ cast operator, which weakens function domains:

```
Definition divide' n := ?d (divide n).
```

When applied, the extracted divide' function checks that its second argument is strictly positive, and raises a runtime cast error otherwise:

```
# divide' 4 2;;
- : nat = 2
# divide' 10 0;;
Exception: Failure "Cast has failed".
```

There are two major challenges to supporting casts of properties: decidability and the potential for cast errors.

Decidability. Since casts lift statically-established properties to runtime checks, one can only apply casts on *decidable* properties. For this, we rely on a Decidable type class, along with an extensible library of decidability instance. (We extend the Coq/HoTT Decidable type class library.) The decision procedure for a given property is synthesized automatically through type class resolution.

For instance, the OCaml code for divide' calls our cast function after applying the `decidable_le_nat` decision procedure for $\leq$:

```
let divide' n m =
  cast_fun_dom (decidable_le_nat 1) (divide n)
```

Cast errors. Notice how the OCaml client code above does not obtain optional values when using divide', but instead obtains plain nats if the cast succeeds, or a runtime error otherwise. To avoid imposing a monadic style to deal with the potential for cast errors, we can represent cast failure in Coq as an *axiom*.[4]

Specifically, we introduce one axiom, failed_cast, which states that for any indexed property on elements of type A, we can build a value of type $\{a{:}A \mid P\ a\}$:

```
Axiom failed_cast :
  ∀ {A:Type} {P : A → Prop} (a:A) (msg: Prop), {a : A | P a}.
```

This axiom is obviously a lie, but it allows us to provide a cast operator as a function of type $A \to \{a{:}A \mid P\ a\}$, even though a cast can fail. Hence programmers can use cast, denoted ?, without having to deal with optional values.

```
Definition cast (A:Type) (P : A → Prop)
  (dec : ∀ a, Decidable (P a)) : A → {a : A | P a} :=
fun a: A ⇒
  match dec a with
    | inl p ⇒ (a ; p)
    | inr _ ⇒ failed_cast a (P a)
  end.
```

The cast operator applies the decision procedure to the given value and, depending on the outcome, returns either the dependent pair with the obtained proof, or a failed_cast. Considering the definition of cast, we see that a cast fails if and only if the property $P\ a$ does not hold according to the decision procedure. A subtlety in the definition of cast is that the casted value must not be exposed as a dependent pair if the decision procedure fails. An alternative definition could always return $(a\ ;\ x)$ where x is some error axiom if the cast failed. Doing so, however, would ruin the interest of the cast framework in the context of program extraction: because all properties (in Prop) are erased, a casted value would not be extracted to a value associated with a runtime check, but just to a plain, unchecked value.

The cast operator denoted $?^d$ weakens the domain of a plain function type $\{a : A \mid P\ a\} \to B$, which expects a value of a subset type as argument, into a standard function type $A \to B$, by downcasting the argument:

```
Definition cast_fun_dom (A B : Type) (P: A → Prop)
  (dec: ∀ a, Decidable (P a)) : ({a : A | P a} → B) → A → B :=
  fun f a ⇒ f (? a).
```

The cast framework includes several other cast operators—some subtleties arise with dependent function types—and can be applied to other rich properties like records (see [6] for details).

3. Recovering Type Dependencies

To recover type dependencies, we propose to exploit *type isomorphisms* between dependently-typed structures and plain structures with subset types, and then exploit the cast framework presented in the previous section to safely eliminate the subset types.

[1] We configure extraction to map the nat inductive type from Coq to OCaml's `int` primitive type, and similarly with list.

[2] We intentionally use the old English form *casted*, and not the currently accepted form *cast*, in order to be able to distinguish a *casted function*, *i.e.* a function that is subject to a cast, from a *cast function*, *i.e.* a function that is used to perform a cast. Similarly, this allows us to distinguish a *casted value*, *i.e.* a value subject to a cast, and a *cast value*, *i.e.* the value of a cast expression.

[3] Note that we use the term *cast* in the standard way, to denote a type assertion with an associated runtime check [4]—this differs from the non-traditional use of *cast* in the Coq reference manual (1.2.10) where it refers to a static type assertion.

[4] Representing cast failure as an axiom is particularly important to support gradual certified programming within Coq [6]. If the focus is only on extraction, then a monadic approach is perfectly reasonable, because the error monad used in Coq can be extracted in a transparent manner to OCaml, such that existing OCaml code does not have to deal with monadic values.

For instance, we can establish an isomorphism between a dependent pair packaging the dependently-typed dstack with its index n and a plain list with a property:

$$\{n{:}\mathsf{nat} \,\&\, \mathsf{dstack}\ n\} \sim \{l{:}\mathsf{list}\ \mathsf{nat} \mid n = \mathsf{length}\ l\}.$$

Similarly, we can establish an isomorphism between dependently-typed instructions dinstr and plain instructions as follows:

$$\{n\ m{:}\mathsf{nat} \,\&\, \mathsf{dinstr}\ n\ m\} \sim \{i{:}\mathsf{instr} \mid \mathsf{valid}\ n\ m\ i\}.$$

where instr is a plain inductive structure for instructions:

```
Inductive instr : Set :=
| IConst : nat → instr
| IPlus : instr.
```

and valid is the (trivial, omitted) relation that embodies the constraints expressed in dinstr. Once the isomorphism is applied, it is possible to get rid of the subset types by applying the cast framework discussed in the previous section. The composition of both techniques allows us to lift exec from a function of type:

$$\forall n\ m : \mathsf{nat},\ \mathsf{dinstr}\ n\ m \to \mathsf{dstack}\ n \to \mathsf{dstack}\ m$$

which relies on both dependently-typed structures dinstr and dstack, to a function of (simple) type:

$$\mathsf{nat} \to \mathsf{nat} \to \mathsf{instr} \to \mathsf{list}\ \mathsf{nat} \to \mathsf{list}\ \mathsf{nat}$$

A further step is to get rid of the first two arguments of exec, which are irrelevant in the extracted code. We can do that by establishing a stronger *dependent isomorphism*, *i.e.* an isomorphism enriched with a functional dependency: a way to recover the index of an indexed structure based on a plain structure. In the case of dstack and list, the n index of dstack can be easily computed from a given list, by taking its length. Also, in a call to exec, the length of the stack is exactly the first index of the corresponding dinstr. Given this first index, and a plain instruction, we can compute the second index. Like property casts, and unlike the computation of the dstack index, computing the second index of a dinstr can fail at runtime, if we discover that the instruction is not compatible with the first index. With these two dependent isomorphisms, we can lift exec to a function of type:

$$\mathsf{instr} \to \mathsf{list}\ \mathsf{nat} \to \mathsf{list}\ \mathsf{nat}$$

which is exactly the expected plain type that an ML programmer would expect. Crucially, the extracted lifted function is safe in that it internally performs the necessary morphing of structures and runtime checks of properties, as needed, before calling the original Coq-extracted function.

The type isomorphisms are similar to those studied in algebraic ornaments [2], where a basic data structure can be augmented with additional logic to form an indexed data type. Here we add the refinement types as a step that allows us to go to dynamic checking of the data logic. The dependent isomorphisms that allow us to hide index arguments from lifting functions seem also novel. Our approach can in fact be characterized as implementing a form of *dependent interoperability* [3] for Coq. Compared to that work, our approach is more general (*e.g.* we can interoperate across type constructors), and is fully (and mechanically) verified.

Availability. The cast framework for Coq is available at:
https://github.com/tabareau/Cocasse.
The dependent type isomorphisms framework is, at the time of this writing, still at a pre-release stage.

References

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

[2] 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.

[3] 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.

[4] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[5] M. Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer-Verlag, 2007.

[6] É. Tanter and N. Tabareau. Gradual certified programming in Coq. In *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*, Pittsburgh, PA, USA, Oct. 2015. ACM Press. To appear. Preprint available on arXiv.