Gradual Certified Programming in Coq

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

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

Abstract

Expressive static typing disciplines are a powerful way to achieve high-quality software. However, the adoption cost of such techniques should not be under-estimated. Just like gradual typing allows for a smooth transition from dynamically-typed to statically-typed programs, it seems desirable to support a gradual path to certified programming. We explore gradual certified programming in Coq, providing the possibility to postpone the proofs of selected properties, and to check "at runtime" whether the properties actually hold. Casts can be integrated with the implicit coercion mechanism of Coq to support implicit cast insertion *à la* gradual typing. Additionally, when extracting Coq functions to mainstream languages, our encoding of casts supports lifting assumed properties into runtime checks. Much to our surprise, it is not necessary to extend Coq in any way to support gradual certified programming. A simple mix of type classes and axioms makes it possible to bring gradual certified programming to Coq in a straightforward manner.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages—Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification Techniques

Keywords Certified programming, refinements, subset types, gradual typing, casts, program extraction, Coq.

1. Introduction

In *Certified Programming with Dependent Types*, Chlipala sketches two main approaches to certified programming [5]. In the classical program verification approach, one separately writes a program, its specification, and the proof that the program meets its specification. A more effective technique is to exploit rich, dependent types to integrate programming, specification and proving into a single phase: specifications are expressed as types, as advocated by Sheard et al. [25] in what they call *language-based verification*. While rich types are a powerful way to achieve high-quality software, we believe that the adoption cost of such techniques is not to be under-estimated. Therefore, it seems desirable to support a *gradual* path to certified programming with rich types, just like gradual typing allows for a smooth transition from dynamically-typed to statically-typed programs [26]. Indeed, the idea of progressively strengthening programs through a form of gradual checking has already been applied to a variety of type disciplines, like typestates [12, 30], information flow typing and security types [8, 9], ownership types [24], annotated type systems [28], and effects [3]. Recent developments like property-based testing for Coq [7] and randomized testing based on refinement types annotations [23] are complementary efforts to make language-based verification more practical and attractive.

In this article, we consider a gradual path to certified programming in Coq, so that programmers can safely postpone providing some proof terms. We focus mostly (but not exclusively) on *subset types*, which are the canonical way to attach a property to a value. 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 $P\ a$.

Constructing a value of type $\{a{:}A \mid P\ a\}$ requires the associated proof term of type $P\ a$. Currently, Coq has two mechanisms to delay providing such a proof term. First, one can use `Program`, a facility that allows automatic coercions to subset types leaving proof obligations to be fulfilled after the definition is completed [27]. This is only a small delay however, because one must discharge all pending obligations before being able to use the defined value. The second mechanism is to *admit* the said property, which makes Coq accept a definition on blind faith, without any proof. This solution is unsatisfactory from a gradual checking point of view, because it is *unsafe*: there is no delayed checking of the unproven property. Therefore, a function that expects a

DLS'15, October 27, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2816707.2816710

value with a given property might end up producing incorrect results. The motto of gradual checking, *trust but verify*, is therefore not supported.

The main contribution of this work is to provide safe casts[1] for Coq, paving the way for gradual certified programming, and to show that this is feasible entirely within standard Coq, without extending the underlying theory and implementation. When casting a value a of type A to the rich type $\{a{:}A \mid P\ a\}$, the property $P\ a$ is checked as needed, forbidding unsafe projection of the value of type A from the dependent pair. Note that because Coq is dependently-typed (ie. types can be dependent arbitrarily on computations and values), there is no rigid compile-time/runtime distinction: therefore, cast errors can possibly occur both as part of standard evaluation (triggered with `Eval`) and as part of type checking, during type conversion.

A key feature of our development is that we support a smooth gradual path to certified programming that avoids imposing a global monadic discipline to handle the possibility of cast errors. Technically, this is achieved thanks to the (possibly controversial) choice of representing cast failures in Coq as an inconsistent axiom, so that failed casts manifest as non-canonical normal forms (*e.g.* a normal form of type bool is either true, false, or a cast failure).

Section 2 provides an informal tour of gradual certified programming with subset types in Coq, through a number of examples. We then dive into the details of the approach, namely type classes for decidability (Section 3) and an axiomatic representation of casts (Section 4). Section 5 then discusses implicit cast insertion *à la* gradual typing. Sections 6 and 7 focus on higher-order casts, with both simple and dependent function types—the latter being subtly more challenging. Section 8 describes the use of casts to protect functions extracted to mainstream languages that do not support subset types. Section 9 briefly describes the main properties of our approach, which follow directly from being entirely developed within standard Coq. Section 10 shows how our approach scales beyond subset types to other dependently-typed constructions, such as record types, and illustrates how it is possible to customize the inference of decision procedures. Section 11 discusses related work and Section 12 concludes.

The code presented in this paper is available as a Coq library at `https://github.com/tabareau/Cocasse`.

2. Gradual Certified Programming in Action

We start by introducing gradual certified programming with subset types through a number of examples of increasing complexity, culminating in a small gradually certified compiler. For now, we only appeal to the intuition of the reader; we discuss the technical details of the approach in Section 3 and beyond.

2.1 First Examples

We now show how casts behave with examples. In this paper, we denote the first and second projections of a pair as $._1$ and $._2$ respectively. First consider a simple definition that is rejected by Coq:

`Definition` n_not_ok : $\{n{:}\mathsf{nat} \mid n < 10\}$:= 5.

This definition is rejected, because the value should be a dependent pair, not just a natural number. Using `Program`,[2] we are left with the obligation to prove that $5 < 10$, which is arguably not too hard.

We could instead use our basic cast operator—denoted ?—to promote 5 to a value of type $\{n{:}\mathsf{nat} \mid n < 10\}$. The semantics is that, if we ever need to evaluate n_good,[3] we will check whether 5 is less than 10:

`Definition` n_good : $\{n{:}\mathsf{nat} \mid n < 10\}$:= ? 5.

`Eval compute in` n_good.

```
= (5; Le.le_n_S 5 9 (...))
: {n : nat | n < 10}
```

We indeed have a dependent pair, whose first component is the number 5 and second component is the proof that $5 < 10$ (elided). We can naturally project the number from the pair:

`Eval compute in` n_good.$_1$.

```
= 5
: nat
```

Of course, we may be mistaken and believe that $15 < 10$:

`Definition` n_bad : $\{n{:}\mathsf{nat} \mid n < 10\}$:= ? 15.

The cast error now manifests whenever we evaluate n_bad:

`Eval compute in` n_bad.

```
= failed_cast 15 (16 <= 10)
: {n : nat | n < 10}
```

Importantly, a failed cast does not manifest as an exception or error, since Coq is a purely functional programming language. Instead, as we will explain further in Section 4, `failed_cast` is a normal form (ie., it cannot be further reduced) of the appropriate subset type, which indicates both the casted value (15) and the violated property ($16 \leq 10$).

Crucially, because n_bad evaluates to a failed cast, we cannot project the natural number, since we do not even have a proper dependent pair:

[1] Note that we use the name "cast" in the standard way [19] to denote a type assertion with an associated runtime check—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.

[2] `Program` is a definition facility that allows automatic coercions to subset types leaving proof obligations to be fulfilled after the definition is completed [27], but before the definition can be used.

[3] Coq does not impose any fixed reduction strategy. Instead, `Eval` is parameterized by a reduction strategy, called a conversion tactic, such as `cbv` (aka. `compute`), `lazy`, `hnf`, `simpl`, etc.

```
Eval compute in n_bad.1.
```

```
= let (a, _) :=
    failed_cast 15 (16 <= 10) in a
: nat
```

At this point, it is worthwhile illustrating a major difference with the use of *admit*, to which we alluded in the introduction. Consider that we use *admit* to lie about 15:

```
Program Definition n_real_bad : {n:nat | n < 10} :=
  15.
Next Obligation. Admitted.
```

In this case, n_real_bad is an actual dependent pair, with the use of *proof_admitted* (an inhabitant of *False*) in the second component:

```
Eval compute in n_real_bad.
```

```
= (15; match proof_admitted return (16 <= 10)
      ...)
: {n : nat | n < 10}
```

This means that we are able to project the number from n_real_bad without revealing the lie:

```
Eval compute in n_real_bad.1.
```

```
= 15
: nat
```

2.2 Casting Lists

Casting a list of elements of type A to a list of elements of type $\{a\colon A \mid P\ a\}$ simply means mapping the cast operator ? over the list. For instance, we can claim that the following list is a list of 3s:

```
Definition list_of_3: list {n:nat | n = 3} :=
  map ? (3 :: 2 :: 1 :: nil).
```

If we force the evaluation of list_of_3, we obtain a list of elements that are either 3 with the proof that 3 = 3, or a failed cast:

```
Eval compute in list_of_3.
```

```
= (3; eq_refl)
   :: failed_cast 2 (2 = 3)
      :: failed_cast 1 (1 = 3) :: nil
: list {n : nat | n = 3}
```

Note the difference between a list of type list $\{a : A \mid P\ a\}$ and a list of type $\{l : \text{list } A \mid P\ l\}$. While the former expresses that each element a of the list satisfies $P\ a$, the latter expresses that the list l as a whole satisfies $P\ l$. Casting works similarly for other inductively-defined structures.

2.3 A Gradually Certified Compiler

We now show how to apply casts to a (slightly) less artificial example. Consider a certified compiler of arithmetic expressions, adapted from Chapter 2 of CPDT [5].

Source language. The source language includes the following binary operations:

```
Inductive binop : Set := Plus | Minus | Times.
```

Expressions are either constants or applications of a binary operation:

```
Inductive exp : Set :=
  | Const : nat → exp
  | Binop : binop → exp → exp → exp.
```

The semantics of binary operations is as expected:

```
Definition evalBinop (b: binop) : nat → nat → nat :=
  match b with
    | Plus ⇒ plus
    | Minus ⇒ sub
    | Times ⇒ mult
  end.
```

So is the semantics of evaluating expressions:

```
Fixpoint evalExp (e: exp) : nat :=
  match e with
    | Const n ⇒ n
    | Binop b e1 e2 ⇒
      (evalBinop b) (evalExp e1) (evalExp e2)
  end.
```

Stack machine. We now introduce the intermediate language of instructions for a stack machine:

```
Inductive instr : Set :=
  | iConst : nat → instr
  | iBinop : binop → instr.
```

A program is a list of instructions, and a stack is a list of natural numbers:

```
Definition prog := list instr.
Definition stack := list nat.
```

Executing an instruction on a given stack produces either a new stack or None if the stack is in an invalid state:

```
Definition runInstr (i: instr) (s: stack): option stack :=
  match i with
    | iConst n ⇒ Some (n :: s)
    | iBinop b ⇒
      match s with
        | arg1 :: arg2 :: s' ⇒
          Some ((evalBinop b) arg1 arg2 :: s')
        | _ ⇒ None
      end
  end.
```

Running a program simply executes each instruction, recursively:

```
Fixpoint runProg (p: prog) (s: stack): option stack :=
  match p with
    | nil ⇒ Some s
    | i :: p' ⇒ match runInstr i s with
```

```
                | None ⇒ None
                | Some s' ⇒ runProg p' s'
              end
  end.
```

Compiler. We now turn to the compiler, which is a recursive function that produces a program given an expression:

```
Fixpoint compile (e: exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒
    compile e1 ++ compile e2 ++ iBinop b :: nil
  end.
```

Hint: there is a bug!

Correct? Of course, one would like to be sure that compile is a *correct* compiler. The traditional way of certifying the compiler is to state and prove a correctness theorem. In CPDT, the compiler correctness is stated as follows:

```
Theorem compile_correct : ∀ (e: exp),
  runProg (compile e) nil = Some (evalExp e :: nil).
```

Namely, executing the program returned by the compiler on an empty stack yields a well-formed stack with one element on top, which is the same value as interpreting the source program directly.

It turns out that the theorem cannot be proven directly by induction on expressions because of the use of nil in the statement of the theorem: the induction hypotheses are not useful. Instead, one has to state a generalized version of the theorem, whose proof does go by induction, and then prove compile_correct as a corollary [5].

Instead of going into such a burden as soon as the compiler is defined, one may want to assert correctness and have it checked dynamically. With our framework, it is possible to simply cast the compiler to a correct compiler. To make the following exposition clearer, we first define what a correct program (for a given source expression) is:

```
Definition correct_prog (e: exp) (p: prog) : Prop :=
  runProg p nil = Some (evalExp e :: nil).
```

To exploit gradual certified programming to claim that compile is correct using a cast, we could try to use our cast operator ?, to attempt to give compile the type $\{f\colon \mathsf{exp} \to \mathsf{prog} \mid \forall\, e{:}\mathsf{exp},\ \mathsf{correct\_prog}\ e\ (f\ e)\}$. This is however undecidable because the property quantifies over all expressions. (In fact, such a cast is rejected by our framework, as discussed in Section 3.) Instead, we need to resort to a higher-order cast operator, denoted ∀?, which can lazily check that the compiler is "apparently" correct by checking that it produces correct programs whenever it is used:

```
Definition correct_comp :=
  ∀ e: exp, {p: prog | correct_prog e p }.

Definition compile_ok : correct_comp := ∀? compile.
```

Let us now exercise compile_ok. The following evaluation succeeds:

```
Eval compute in
  compile_ok (Binop Plus (Const 2) (Const 2)).

= (iConst 2 :: iConst 2 :: iBinop Plus :: nil;
   eq_refl)
: {p : prog | correct_prog ...}
```

However, the cast fails when using a (non-commutative!) subtraction operation:

```
Eval compute in
  compile_ok (Binop Minus (Const 2) (Const 1)).

= failed_cast (iConst 2 :: iConst 1
                   :: iBinop Minus :: nil)
         (Some (0 :: nil) = Some (1 :: nil))
: {p : prog | correct_prog ...}
```

Indeed, the compiler incorrectly compiles application nodes, compiling sub-expressions in the wrong order! The last argument of `failed_cast`—the invalid property—is explicit about what went wrong: the compiler produced a program that returns 0, while the interpreter returned 1.

Finally, suppose we write a runc function that requires a correct compiler as argument:

```
Definition runc (c: correct_comp) (e: exp) :=
  runProg (c e).1 nil.
```

We can use the cast framework to pass compile as argument, but in case the compiler behaves badly, runc fails because it cannot apply the projection $._1$ to a failed cast:

```
Eval compute in runc (∀? compile)
                 (Binop Minus (Const 2) (Const 1)).

= ...
  (let (a, _) := failed_cast
    (Some (0 :: nil) = Some (1 :: nil)) ...
: option stack
```

Again, note that if we had used *admit* to lie about compile, then runc would not have detected the violation of the property, and would have therefore returned an incorrect result.

3. Casts and Decidability

What exactly does it mean to *cast* a value a of type A to a value of the rich type $\{a : A \mid P\ a\}$? There are two challenges to be addressed. First, because we are talking about *safe* casts, it must be possible to check, for a given a, whether $P\ a$ holds. This means that $P\ a$ must be *decidable*. Second, because it may be the case that $P\ a$ does not hold, we must consider how to represent such a "cast error", considering that Coq does not have any built-in exception mechanism. For decidability, we exploit the type class mechanism of Coq, as explained in this section. For failed casts, we exploit axioms (Section 4).

3.1 Decidable Properties

The Decidable type class, which is used in the Coq/HoTT library[4], is a way to characterize properties that are decidable. To establish that a property is decidable, one must provide an explicit proof that it either holds or not:

```
Class Decidable (P : Prop) := dec : P + ¬ P .
```

Note that the disjunction is encoded using a sum type (+, which is in Type) instead of a propositional disjunction (∨, which is in Prop) in order to support projecting the underlying proof term and use it computationally as a decision procedure for the property.[5]

The Coq type class system can automatically infer the decision procedure of a complex property, using type class resolution, when a cast is performed. For that, the appropriate generic decidability instances must be provided first, but those instances are implemented only once and are already part of the Decidable library or can be added as needed. For example, the following instance definition (definition omitted) allows Coq to infer decidability—and build the associated decision procedure—for a conjunction of two decidable properties by evaluating the decision procedure for each property:

```
Instance Decidable_and (P Q: Prop) (HP : Decidable P)
         (HQ : Decidable Q) : Decidable (P ∧ Q).
```

Also, whenever a proposition has been proven, it is obviously decidable (inl is the left injection on a sum type):

```
Instance Decidable_proven (P : Prop) (ev : P):
  Decidable P := inl ev.
```

This instance allows programmers to mix proven and decidable properties, for instance by inferring that $P \wedge Q$ is decidable if P is decidable and Q is proven.

Another interesting instance is the one exploits the fact that every property that is equivalent to a decidable property is decidable:

```
Definition Decidable_equivalent {P P' : Prop}
    (HPP' : P' ↔ P) `{Decidable P'} : Decidable P.
```

We will exploit this instance in Section 10 to synthesize more efficient decision procedures.

If type class resolution cannot find an instance of the Decidable class for a given property, then casting to a subset type with that property fails statically. This happens if we try to cast compile directly to a function subset type with a universally-quantified property, as discussed in Section 2.

3.2 Leveraging Type Class Resolution

Depending on the structure of the property to be established, we can get decidability entirely for free. In fact, in the compiler example (Section 2.3), the decidability of correct_prog was automatically inferred! We now explain how this automation was achieved.

The correct_prog property is about equality of the results of running programs, which are option stacks, or more explicitly, option list nats. The Decidable type class already allows, with its instances, to automatically obtain complex correct decision procedures based on composition of atomic ones (Sect. 3). For correct_prog to enjoy this full automation, the Decidable library needs to include instances that allow equality of lists and options to be inferred. More precisely, we provide a type class for decidable equality, $\text{Decidable}_=$:[6]

```
Class Decidable= (A : Type) :=
  { eq_dec : ∀ a b : A, Decidable (a = b) }.
```

Based on this decidable equality class, we can define once and for all how to derive the decidability of the equality between lists of A or options of A provided that equality is decidable for A:

```
Instance Decidable_eq_list : ∀ A (HA: Decidable= A)
  (l l': list A), Decidable (l = l').

Instance Decidable_eq_option : ∀ A (HA: Decidable= A)
  (o o': option A), Decidable (o = o').
```

By also declaring the corresponding $\text{Decidable}_=$ instances for the list and option type constructors, the type class resolution mechanism of Coq is able to automatically build the correct decision procedures for properties that state equality between arbitrary nestings of these type constructors, such as correct_prog. A well-furnished decidability library allows developers to seamlessly enjoy the benefits of gradual certified programming.

We come back to decidability in Section 10, when describing casts on rich records, in order to show how one can specialize the decision procedure to use in specific cases, for instance to obtain a procedure that is more efficient than the default one.

4. Casts and Axioms

Intuitively, the basic cast operator ? should be defined as a function cast of type $A \rightarrow \{a : A \mid P\ a\}$ (assuming that $P\ a$ is decidable). To perform such a cast implies exploiting the decidability of $P\ a$: checking (and hence evaluating) whether $P\ a$ holds or not. If it holds true, the cast succeeds. The cast function can simply return the dependent pair with the value a and the proof. If $P\ a$ does not hold, the cast fails. How should such errors manifest?

[4] `https://github.com/HoTT/HoTT`

[5] An equivalent decision procedure mechanism is implemented in the Ssreflect library [13], using boolean reflection. We discuss the differences between the two approaches in Appendix A. It must be noticed already that the differences are minor and our cast mechanism works perfectly well with both ways of formalizing decidability.

[6] A similar type class is also used in the Coq/HoTT library under the name DecidablePaths.

4.1 The Monadic Approach

The traditional way to support errors in a purely functional setting is to adopt a monadic style. For instance, we could define cast to return option $\{a{:}A \mid P\ a\}$ instead of just $\{a{:}A \mid P\ a\}$. Then, a cast failure would simply manifest as None. This is all well and understood, but has serious consequences from a software engineering point of view: it forces all code that (potentially) uses casts to also be written in monadic style. Because the philosophy of gradual typing entails that casts may be added (or removed) anywhere as the software evolves, it means that the entire development has to be defensively written in monadic style. For instance, consider the definition of runc in Section 2.3:

```
Definition runc (c: correct_comp) (e: exp) :=
  runProg (c e).1 nil.
```

If it were possible to check eagerly that compile is correct, the monadic cast would produce a value of type option correct_comp, and the client calling runc (? compile) would simply have to locally deal with the potential of failure. However, since correct_comp is undecidable, the only solution is to delay casts, which means that the casted compiler would now have type $\forall\ e : \mathsf{exp}$, option $\{p : prog \mid$ correct_prog $e\ p\}$. This in turn implies that *all users* of the compiler (such as runc) have to be prepared to deal with optional values. The argument type of runc would have to be changed, and its body as well because ($c\ e$) would now return an option correct_prog, not a correct_prog. This non-local impact of deciding to statically establish guarantees or defer them to runtime is contrary to the smooth transition path that gradual typing is meant to support.

After all, every practical functional programming language does some compromise with purity[7], supporting side effects like references and exceptions directly in the language, rather than through an explicit monadic encoding. The upside of sacrificing purity is that these side effecting operations can be used "transparently", without having to adopt a rigid discipline like monads, which—despite various improvements such as [22]—is still not free from software engineering challenges. So, what does it mean to embed cast errors in such a transparent manner in Coq?

4.2 The Axiomatic Approach

We introduce a novel use of axioms, not to represent what is assumed to be true, but to represent errors. This allows us to provide the cast operator as a function of type $A \to \{a : A \mid P\ a\}$. 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 $\{a : A \mid P\ a\}$: [8]

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

Obviously, *failed_cast* is a lie. This lie is used in the definition of the cast operator, in case the decision procedure indicates that the property does not hold:

```
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. Our definition has the advantage of reporting a cast failure as soon as the casted value is used (even though the property attached to it is not).[9]

We introduce the ? notation for cast, asking Coq to infer the property and the evidence of its decidability from the context:

```
Notation "?" := (cast _ _ _).
```

4.3 Heresy!

Using an axiom to represent failed casts is (slightly!) heretical from a theoretical viewpoint. As a matter of fact, one can use a cast to inhabit False, for instance by pretending that 0 comes with a proof of False and then projecting the second component:

```
Definition unsound : False := (? 0).2.
```

In this sense, the monadic approach is preferrable, as it preserves consistency. However, the axiomatic approach is an interesting alternative to using plain axioms and admitted definitions in Coq—which are, after all, the only pragmatic solutions available to a Coq practitioner who does not want to wrestle with a given proof immediately. Axiomatic casts are superior in many ways:

- As discussed above, we cannot project the value component of a subset type with a failed cast (recall that using *admit* provides no such guarantee).

[7] Even Haskell has impure features such as `undefined`, `unsafeCoerce` and `unsafePerformIO`, for pragmatic reasons.

[8] We declare the two first arguments of *failed_cast* as implicit (between {}), and only leave the value a and the *msg* argument as explicit. The argument *msg* is apparently redundant, since it is just defined as $P\ a$ in cast; however, declaring it as an explicit argument together with a allows for clear and concise error messages when cast fails, reporting the violated property for a given value, as illustrated in Section 2.

[9] Appendix B briefly discusses the interplay of evaluation regimes and the representation of cast failures as non-canonical normal forms.

- When things go right (i.e. the cast succeeds), there is no axiom or admitted definition that will block type conversion and evaluation.
- Statically establishing a property or using a cast does not affect the types involved, so the programmer can seamlessly navigate the gradual checking spectrum without having to perform non-local refactorings.

All in all, both the monadic and axiomatic approaches to gradual verification are feasible, and are likely to please different crowds. In fact, we have implemented both approaches in the Cocasse library. In this paper we focus on the axiomatic approach, because of its disruptive potential and software engineering benefits. We believe this approach will be more appealing to pragmatic practitioners who are willing to compromise consistency to some extent in order to enjoy a smooth gradual verification environment. Also, as we discuss in Section 12, there are alternatives to be explored to make the axiomatic approach less heretical.

5. Implicit Casts

The major technical challenge addressed in this work is to provide casts for subset types within Coq. These casts have to be *explicitly* placed by programmers, much like in the seminal work of Abadi *et al.* on integrating static and dynamic typing [1], or in the gradual information flow type system proposed by Disney and Flanagan [8]. Gradual typing is however generally associated with a mechanism of *implicit* cast insertion: the source language, which may not even feature explicit casts, is translated to an internal language with explicit casts [26].

It is possible to achieve implicit cast insertion in Coq by exploiting the *implicit coercion* mechanism.[10]

Implicit coercions in a nutshell. Let us first briefly illustrate implicit coercions in Coq. Assume a decidable indexed property on nat, which is used to define a type rich_nat:

```
Variable P : nat → Prop.
Variable P_dec : ∀ n:nat, Decidable (P n).
Definition rich_nat := {n: nat | P n}.
```

To define an implicit coercion from values of type rich_nat to nat, we first define a function with the appropriate type, and then declare it as an implicit Coercion: Definition rnat_to_nat : rich_nat → nat :=

```
    fun n ⇒ n.1.
Coercion rnat_to_nat : rich_nat ↣ nat.
```

We can now pass a rich_nat to a function that expects a nat, without having to explicitly apply the coercion function:

```
Variable f : nat → nat.
Variable s : rich_nat.
Check f s.
```

[10] https://coq.inria.fr/distrib/current/refman/Reference-Manual021.html

Implicit cast insertion. In order to implicitly insert casts, it is enough to define a standard implicit coercion based on a function that introduces casts. For instance, we define an implicit coercion (cast insertion) from nat to rich_nat:

```
Definition nat_to_rnat : nat → rich_nat := ?.
Coercion nat_to_rnat : nat ↣ rich_nat.
```

Calling a function that expects a rich_nat with a nat argument is now type-correct. Under the hood, the implicit coercion takes care of inserting the cast:

```
Variable g : rich_nat → nat.
Variable n: nat.
Check g n.
```

Compared to standard gradual typing, the limitation of this approach is that Coq does not support universal coercions, so one needs to explicitly define the specific coercions that are permitted. This is arguably less convenient than a general implicit cast insertion mechanism, but it is also more controlled. Because types are so central to Coq programming, it is unclear whether general implicit cast insertion would be useful and not an endless source of confusion. Actually, even in gradually-typed languages with much less powerful type systems, it has been argued that a mechanism to control implicit cast insertion is important [2]. We believe that the implicit coercion mechanism of Coq combined with casts might be a good tradeoff in practice.

6. Higher-Order Casts, Simply

We now consider cast operators for functions. As expected, function casts are enforced lazily similarly to higher-order contracts [10]. We first focus on non-dependent function types of the form $A \to B$. One could want to either strengthen the range of the function, claiming that the return type is $\{b : B \mid P\ b\}$, or vice-versa, to hide the fact that a function expects rich arguments of type $\{a : A \mid P\ a\}$.

6.1 Strengthening the Range

The cast_fun_range operator below takes a function of type $A \to B$ and returns a function of type $A \to \{b : B \mid P\ b\}$. It simply casts the return value to the expected subset type:

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

Example. We can cast a nat → nat function such as S (successor) to a function type that ensures the returned value is less than 10:

```
Definition top_succ : nat → {n:nat | n < 10} := →? S.
```

Then, as expected:

```
Eval compute in top_succ 6.
```

```
= (7; Le.le_n_S 7 9 ...)
: {n : nat | n < 10}
```

And:

```
Eval compute in top_succ 9.
```

```
= failed_cast 10 (11 <= 10)
: {n : nat | n < 10}
```

6.2 Weakening the Domain

Similarly, cast_fun_dom turns a function of type $\{a : A \mid P\ a\} \rightarrow B$, which expects a value of a subset type, into a standard function of type $A \rightarrow B$, by casting the argument to the expected subset type:

```
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).
Notation "?→" := (cast_fun_dom _ _ _ _).
```

Example. The standard division function on natural numbers in Coq, div, is total and pure, but incorrect: when the divisor is 0, the result is 0. We can use subset types to define a pure and correct version, divide, which is total on a restricted domain, by requiring its second argument to be strictly positive:

```
Definition divide: nat → {n: nat | n > 0} → nat :=
  fun a b ⇒ div a b.1.
```

Using this function now forces the programmer to provide a proof that the second argument is strictly positive. This can be achieved with the standard cast operator ?. Alternatively, we can cast divide into a function that accepts plain nats, but internally casts the second argument to ensure it is strictly positive:

```
Definition divide': nat → nat → nat :=
  fun a ⇒ ?→ (divide a).
```

As expected, applying divide' with 0 as second argument produces a cast failure.

```
Eval compute in divide' 1 0.
```

```
= match (let (a, _) := failed_cast 0 (1 <= 0) ...
: nat
```

Arguably, it is more correct for division by zero to manifest as a failure than to silently returning 0. We will also see in Section 8 that weakening the domain of a function is helpful when extracting it to a target language that does not support subset types, because the assumptions expressed in the richly-typed world translate into runtime checks.

7. Higher-Order Casts, Dependently

The higher-order cast operators defined above are not applicable when the target function type is dependently-typed. Recall that in Coq, a dependently-typed function has a type of the form $\forall\ a{:}\ A,\ B\ a$, meaning that the type of the result $(B\ a)$ can depend on the value of the argument a.

For instance, in Section 2.3, we cast compile to the dependent function type correct_comp, which is an alias for the type $\forall\ e{:}$ exp, $\{p{:}\ prog \mid$ correct_prog $e\ p\}$. An alternative would have been to downcast runc, which expects a correct compiler, to a looser function type that accepts any compiler (similarly to what we have done above with divide). We now discuss both forms of casts; as it turns out, weakening the domain of a dependently-typed function is a bit of a challenge.

7.1 Strengthening the Range

Strengthening a function type so that it returns a rich dependent type is not more complex than with a simply-typed function; it just brings the possibility that the claimed property on the returned value also depends on the argument:

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

Examples. We can cast a nat → nat function to a dependently-typed function that guarantees that it always returns a value that is greater than or equal to its argument:

```
Definition f_inc :
  (nat → nat) → ∀ n : nat, {m:nat | (n ≤ m)} := ∀?.
```

Then, as expected:

```
Eval compute in f_inc S 3.
```

```
= (4; Le.le_n_S 2 3 ...)
: {m : nat | 3 <= m}
```

And:

```
Eval compute in f_inc (fun _ ⇒ O) 3.
```

```
= failed_cast 0 (3 <= 0)
: {m : nat | 3 <= m}
```

The above example casts a simply-typed function to a dependently-typed function, also illustrating the binary property $P\ a\ b$ in the range. In the following example, the casted function is dependently-typed. Consider the inductive type of length-indexed lists of nat, and the dependently-typed constructor build_list:

```
Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, nat → ilist n → ilist (S n).
Fixpoint build_list (n: nat) : ilist n :=
 match n with
   | O ⇒ Nil
   | S m ⇒ Cons _ O (build_list m)
 end.
```

We can cast build_list (of type $\forall$ n:nat, ilist n) to a function type that additionally guarantees that the produced list is not empty.

Definition non_empty_build:
  $\forall$ n:nat, {_: ilist n | $n > 0$ } := ∀? build_list.

Then, as expected:

Eval compute in non_empty_build 2.

```
= (Cons 1 0 (Cons 0 0 Nil); ...)
: {_ : ilist 2 | 2 > 0}
```

And:

Eval compute in non_empty_build 0.

```
= failed_cast Nil (1 <= 0)
: {_ : ilist 0 | 0 > 0}
```

7.2 Weakening the Domain

Consider a function that expects an argument of a subset type $\{a : A \mid P\ a\}$, and whose return type depends on the value component of the dependent pair. Such a function has type $\forall\ x\colon \{a : A \mid P\ a\}, B\ x._1$. Weakening the domain in this case means casting this function to the dependent type $\forall\ a : A, B\ a$.

Notably, defining such a cast operator leads to an interesting insight regarding casts in a dependently-typed language. Because cast hides a lie about a value, when casting the argument of a dependently-typed function, the lie percolates at the type level due to the dependency. Consider the intuitive definition of cast_forall_dom, which simply applies cast to the argument:

Definition cast_forall_dom (A: Type) (P: $A \to$ Prop)
    (B: $A \to$ Type) (dec: $\forall$ a, Decidable (P a)) :
  ($\forall$ x: $\{a : A \mid P\ a\}$, $B\ x._1$) $\to$ ($\forall$ $a : A$, B a) :=
  fun f $a \Rightarrow f$ (? a).

Coq (rightfully) complains that:

```
The term "f (? a)" has type "B (? a).1"
while it is expected to have type "B a".
```

Indeed, the return type of the casted function can depend on the argument, yet we are lying about the argument by claiming that it has the subset type $\{a : A \mid P\ a\}$. Therefore, in all honesty, the only thing we know about f (? a) is that it has type $B\ a$ *only if the cast succeeds*—in which case (? a)$._1 = a$. But the cast may fail, in which case ? a is not a dependent pair and (? a)$._1$ cannot be reduced: it is a cast error at the type level.

What can we do about this? We know that cast errors can occur, but we do not want to pollute all types with that uncertainty. Following the axiomatic approach to casts, we can introduce a second axiom, *failed_cast_proj1*, to *hide* the fact that cast errors can occur at the type level. Note that we do not want to pose the equality (? a)$._1 = a$ as an axiom, otherwise we would be relying on the axiom even though the cast succeeds. The axiom is required only to *pretend* that the first projection of a failed cast is actually the casted value[11]:

Axiom *failed_cast_proj1* :
  $\forall$ {A:Type} {$P : A \to$ Prop} {a: A} (msg:Prop),
   (*failed_cast* (P:=P) a msg)$._1 = a$.

Using this axiom allows us to define an operator to hide casts from types, hide_cast_proj1 (notation [?]), as follows:[12]

Definition hide_cast_proj1 (A: Type) (P: $A \to$ Prop)
  (B: $A \to$ Type) (dec: $\forall$ a, Decidable (P a)) (a:A):
  B (? a)$._1 \to B\ a$.
Proof.
  unfold cast. case (dec a); intro p.
  - exact (fun $b \Rightarrow b$).
  - exact (fun $b \Rightarrow$ eq_rect _ _ b _
        (*failed_cast_proj1* (P a))).
Defined.

Notation "[?]" := (hide_cast_proj1 _ _ _ _ _).

The equality coming from *failed_cast_proj1* is necessary to transform the term b of type B (*failed_cast* _ P a msg)$._1$ to a term of type $B\ a$. This is done using the elimination rule eq_rect of the equality type. Here again, we can see that a *failed_cast_proj1* error will only occur if the property $P\ a$ does not hold.

We can now define cast_forall_dom as expected, by adding the hiding of the cast in the return type:

Definition cast_forall_dom (A: Type) (P: $A \to$ Prop)
    (B: $A \to$ Type) (dec: $\forall$ a, Decidable (P a)) :
  ($\forall$ x: $\{a : A \mid P\ a\}$, $B\ x._1$) $\to$ ($\forall$ $a : A$, B a) :=
  fun f $a \Rightarrow$ [?] (f (? a)).
Notation "?∀" := (cast_forall_dom _ _ _ _).

Example. Recall the length-indexed lists of Sect. 7.1. Consider the following dependently-typed function with a rich domain type, which specifies that given a strictly positive nat, it returns an ilist of that length:

Definition build_pos : $\forall$ x: {n: nat | $n > 0$ }, ilist ($x._1$) :=
  fun $n \Rightarrow$ build_list ($n._1$).

We can use ?∀ to safely hide the requirement that $n > 0$:

Definition build_pos' : $\forall$ n: nat, ilist n := ?∀ build_pos.

Then, as expected:

Eval compute in build_pos' 2.

```
= Cons 1 0 (Cons 0 0 Nil)
: ilist 2
```

[11] The key word in the sentence is *pretend*: the new axiom does *not* allow one to actually project a value out of a failed cast; it only serves to hide the potential for cast failure from the types.

[12] This time, we use tactics to define hide_cast_proj1, instead of giving the functional term explicitly as we did for cast. The reason is that because of the dependency, a simple pattern matching does not suffice and extra type annotations have to be added to match in order to help Coq typecheck the dependent pattern matching.

And we can now see *failed_cast_proj1* appearing:

Eval compute in build_pos' 0.

```
= eq_rect ...
 ((fix build_list (n : nat) : ilist n := ...)
  (let (a, _) := failed_cast 0 (1 <= 0) in a))
 0 (failed_cast_proj1 (1 <= 0))
: ilist 0
```

8. Extraction

An interesting feature of Coq in terms of bridging certified programming with practical developments is the possibility to *extract* definitions to mainstream languages. The standard distribution of Coq supports extraction to Ocaml, Haskell, and Scheme; and there exists several experimental projects for extracting Coq to other languages like Scala and Erlang.

Coq establishes a strong distinction between programs (in Type), which have computational content, and proofs (in Prop), which are devoid of computational meaning and are therefore erased during extraction. This allows for extracted programs to be efficient and not carry around the burden of unnecessary proof terms. However, this erasure of proofs also means that subset types are extracted to plain types, without any safeguards. It also means that the use of admitted properties is simply and unsafely erased!

To address these issues, we can exploit our cast framework. By establishing a bridge between properties and computation, casts are extracted as runtime checks, and cast failures manifest as runtime exceptions—which is exactly how standard casts work in mainstream programming languages. This ensures that the assumptions made by certified components extracted to a mainstream language are dynamically enforced.

Example. Recall from Section 6.2 the divide function of type $\mathtt{nat} \to \{n\colon \mathtt{nat} \mid n > 0\} \to \mathtt{nat}$. 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 to a mainstream programming language, because the extracted function has the plain type nat→nat→nat:

Definition divide: $\mathtt{nat} \to \{n\colon \mathtt{nat} \mid n > 0\} \to \mathtt{nat}$:=
  fun a b ⇒ div a b.$_1$.

Extraction *Language Ocaml*.
Extraction divide.

```
let divide a b = div a b
```

The dependent pair corresponding to the subset type has been erased, and divide does not check that the second argument is positive (we extract nat to OCaml's `int`):

```
# divide 1 0;;
- : int = 0
```

If we instead first cast divide to the divide' function with plain type nat→nat→nat, and then extract divide':

Definition divide': nat → nat → nat :=
  fun a ⇒ ?→ (divide a).

Extraction divide'.

```
let divide' a =
  cast_fun_dom (decidable_le_nat 1) (divide a)
```

The inserted cast translates to a runtime check in the extracted code, whose failure results in a runtime cast error:

```
# divide' 1 0;;
Exception: Failure "Cast has failed".
```

Extracting axioms as exceptions. By default, the use of an axiom translates to a runtime exception in Ocaml. In order to make the error message more informative, we explicitly instruct Coq to extract uses of *failed_cast* as follows:[13]

Extract *Constant failed_cast* ⇒
  "failwith ""Cast has failed""".

Appendix B, which discusses evaluation regimes, includes discussion about some subtleties that arise when extracting to an eager language like Scheme or Ocaml.

Finally, note that the second axiom we introduced in Section 7.2, *failed_cast_proj1*, does not need to be extracted at all: it is used to convert two types that are equal after extraction (because they only differ in propositional content).

9. Properties

The development of gradual checking of subset types we have presented is entirely internalized in Coq: we have neither extended the underlying theory nor modified the implementation. The only peculiarities are the use of the *failed_cast* and *failed_cast_proj1* axioms. As a consequence, a number of properties come "for free".

Canonicity. Coq without axioms enjoys a canonicity property, which states that all normal forms correspond to canonical forms. For instance, all normal forms of type `bool` are either `true` or `false`.

Cast errors. Canonicity is only violated by the use of axioms. Here, this means that the only non-canonical normal forms are terms with *failed_cast* (or *failed_cast_proj1*) inside. More precisely, a cast failure in Coq is any term t such that $t = E[\mathtt{failed\_cast}\ v\ p]$, where v is the casted value and p is a false property (ditto for *failed_cast_proj1*). In Coq, for cast errors that manifest at the value level, the evaluation context E is determined by the evaluation regime specified when calling Eval. For cast errors that manifest at the type

[13] To be more helpful in the error reporting, we do provide a string representation of the casted value by using a showable type class, similar to `Show` in Haskell (see code in the distribution). However, we cannot provide the information of the violated property, because there is currently no way to obtain the string representation of an arbitrary Prop within Coq.

level, E follows the reduction strategy for type conversion, which is coarsely a head normal-form evaluation with optimization for constants.

Soundness via extraction. The canonicity of Coq and the definition of cast errors, together with the assumption that program extraction in Coq is correct (and axioms are extracted as runtime errors), entail the typical type soundness property for gradually-typed programs, i.e. programs with safe runtime casts [16, 26]: the only stuck terms at runtime are cast errors.[14]

Termination of casts. Because decision procedures are defined within Coq, casts are guaranteed to terminate. This is in contrast to some approaches, like hybrid type checking in Sage [15, 17], in which decision procedures are defined within a language for which termination is not guaranteed.

Simplification at extraction. Because propositions are erased at extraction, the *failed_cast_proj1* axiom is never extracted in the target language and thus cannot fail. This means that in the extracted program, hide_cast_proj1 is always extracted to the identity function, and errors can only manifest through the *failed_cast* axiom.

10. Casting More Dependent Types: Records

Until now, we have developed the axiomatic approach to gradual verification in Coq with subset types, because they are the canonical way to attach a property to a value. However, the approach is not specific to subset types and accomodates other dependently-typed structures commonly used by Coq developers, such as *record types*. To stress that our approach is not restricted to subset types, we now illustrate how to deal with dependent records. We also use this example as a case study in customizing the synthesis of correct decision procedures through the Decidable type class.

Rationals. Consider the prototypical example provided in the Coq reference manual[15]: a record type for rational numbers, which embeds the property that the divisor is not zero, and that the fraction is irreducible. The type of rational numbers, with their properties, is defined as:

```
Record Rat : Set := mkRat
  {sign : bool;
   top : nat;
   bottom : nat;
   Rat_bottom_cond : 0 ≠ bottom;
   Rat_irred_cond : ∀ x y z,
       y × x = top ∧ z × x = bottom → 1 = x}.
```

[14] Note that if the target language is impure, then it is possible to break the safety of program extraction altogether (eg. by passing an impure Ocaml function as input to a Coq-extracted function). This general issue is independent of casting and beyond the scope of this work. Ensuring safe interoperability between a purely functional dependently-typed language like Coq and a language with impure features is a challenging research venue.

[15] `https://coq.inria.fr/refman/Reference-Manual004.html#sec61`

Casting rationals. The property Rat_bottom_cond is obviously decidable. It is less clear for the property Rat_irred_cond, which uses universal quantification. Indeed, in general, there is no decision procedure for a universally-quantified decidable property over natural numbers, because the set of natural numbers is infinite. So it seems we cannot use the cast framework to create rationals without having to provide proofs of their associated properties.

Interestingly, it *is* possible to use casts for rationals despite the fact that Rat_irred_cond cannot be directly declared to be decidable. We review three different approaches in this section. They all exploit the fact that if we can prove that a decidable property is equivalent to Rat_irred_cond, then Rat_irred_cond is decidable (Section 3).

We define a cast operator for Rat, which takes the three values for sign, top and bottom, two (implicitly-passed) decision procedures *dec_rat_bottom* and *dec_rat_irred*, and checks the two properties:

```
Definition cast_Rat (s:bool) (t b: nat)
  {dec_rat_bottom : Decidable _}
  {dec_rat_irred : (0 ≠ b) → Decidable _} : Rat :=
  match dec _ with
    | inl Hb ⇒
      match dec (Decidable := dec_rat_irred Hb) _ with
        | inl Hi ⇒ mkRat s t b Hb Hi
        | _ ⇒ failed_cast_Rat s t b
      end
    | _ ⇒ failed_cast_Rat s t b
  end.
```

As before, the definition of the cast operator appeals to an inconsistent axiom in the case a property is violated. The *failed_cast_Rat* axiom states that any three values are adequate to make up a Rat:[16]

```
Axiom failed_cast_Rat : ∀ (s:bool) (t b: nat), Rat.
```

Note that we use a type dependency in cast_Rat to allow the decision procedure of *dec_rat_irred* to use the fact that Rat_bottom_cond holds in the branch where it is used.

A decision procedure based on bounded quantification. A first approach to establish a decision procedure for irreducibility is to exploit that it is equivalent to the same property that quantifies over *bounded* natural numbers. We first define the type of bounded naturals (and we introduce an implicit coercion from bnat to nat):

```
Definition bnat (n:nat) := {m : nat | m ≤ n}.
```

[16] It is necessary to define custom axioms and cast operators for each new record type. This limitation was not apparent with subset types, because they are a general purpose structure, while records are specific. To limit the burden of adoption, it would be interesting to define a Coq plugin that automatically generates the axioms and cast operators (whose definitions are quite straightforward).

and define a general instance of Decidable, which allows building a decision procedure for any universally-quantified property over bounded naturals:

```
Instance Decidable_forall_bounded k
  (P:bnat k→Prop) (HP : ∀ n, Decidable (P n)) :
  Decidable (∀ n, P n).
```

We can then establish how to synthesize a decision procedure for Rat_irred_cond by establishing that it is equivalent to a similar property, where the quantification is bounded by the max of top and bottom:

```
Definition Rat_irred_cond_bounded top bottom ‘(0 ≠ bottom):
  (∀ x y z: bnat (max top bottom),
     y × x = top ∧ z × x = bottom → 1 = x) ↔
  (∀ x y z: nat, y × x = top ∧ z × x = bottom → 1 = x).
```

Note that it is crucial to be able to use the fact that $0 \neq$ bottom holds in the proof of equivalence, as it simply does not hold when bottom = 0.

Then, the Decidable instance for Rat_irred_cond is simply defined by connecting it to the bounded property through the Decidable_equivalent instance:

```
Instance Rat_irred_cond_dec_bounded top bottom
          ‘(0 ≠ bottom) : Decidable _ :=
  Decidable_equivalent
    (Rat_irred_cond_bounded top bottom H).
```

Example. It is now possible to define a rational number without having to prove the two side conditions.

```
Definition Rat_good := cast_Rat true 5 6.
Eval compute in top Rat_good.
```

```
= 5
: nat
```

Exactly in the same way as the first projection of a dependent pair cannot be recovered if the cast fails, sign, top or bottom can not be recovered if cast_Rat fails.

```
Definition Rat_bad := cast_Rat true 5 10.
Eval compute in top Rat_bad.
```

```
= let (_, top, bottom, _, _) :=
      failed_cast_Rat true 5 10  in top
: nat
```

Note that the evaluation of top Rat_bad takes a significant amount of time, because the decision procedure involves checking every possible x y z: bnat 10, which amounts to checking more than 1000 properties. Indeed, a simple cast as above takes around 2 seconds on a recent computer.

We now show that we can improve the cast on rational numbers by using more efficient decision procedures over equivalent properties.

A decision procedure using binary natural numbers. In the Coq standard library, there is a *binary* representation of integers, Z, which is much more efficient but less easy to reason about. We can exploit this representation by showing that the property Rat_irred_cond in Z implies the property in nat:

```
Definition Rat_irred_cond_Z top bottom ‘(0 ≠ bottom):
  (∀ x y z: bnat (max top bottom),
     Z.mul y x = top ∧ Z.mul z x = bottom → 1 = x) ↔
  (∀ x y z: nat, y × x = top ∧ z × x = bottom → 1 = x).
Instance Rat_irred_cond_dec top bottom ‘(0 ≠ bottom):
  Decidable _ :=
  Decidable_equivalent
    (Rat_irred_cond_Z top bottom H).
```

In this manner, the time for evaluating the same "bad" rational number cast as Rat_bad decreases by a factor of 10!

A decision procedure based on gcd. We can go even one step further and avoid doing an exhaustive (even if finite) check: the property Rat_irred_cond is actually equivalent to the gcd of top and bottom being equal to 1:

```
Definition Rat_irred_cond_gcd top bottom ‘(0 ≠ bottom) :
  (Z.gcd (top:nat) bottom = 1) ↔
  (∀ x y z, y × x = top ∧ z × x = bottom → 1 = x).
Instance Rat_irred_cond_gcd_dec top bottom
          (Hbot : 0 ≠ bottom) : Decidable _ :=
  Decidable_equivalent
    (Rat_irred_cond_gcd top bottom Hbot).
```

Computing the same bad cast is now instantaneous.

11. Related Work

There is plenty of work on rich types like refinement types [4, 11, 20, 31] (which roughly correspond to the subset types of Coq [27]), focusing mostly on how to maintain statically decidable checking (eg. through SMT solvers) while offering a refinement logic as expressive as possible. Liquid types [20], and their subsequent improvements [6, 29], are one of the most salient example of this line of work. Of course, to remain statically decidable, the refinement logics are necessarily less expressive than higher-order logics such as Coq and Agda. In this work we focus on Coq, giving up fully automatic verification. This being said, Coq allows a mix of automatic and manual theorem proving, and we extend this combination with the possibility to lift proofs of decidable properties to delayed checks with casts. Notably, the set (and shape) of decidable properties is not hardwired in the language, but is derived from an extensible library. We believe our approach is applicable to Agda as well, since the main elements (axioms and type classes) are also supported in Agda. However, the devil is certainly in the details.

Interestingly, Seidel *et al.* recently developed an approach called type targeted testing to exploit refinement type annotations not for static checking, but for randomized property-based testing [23]. This supports a progressive approach by which programmers can first enjoy some benefits of (unchecked) refinement type annotations for testing, and then eventually turn to full static checking when they desire. While the authors informally qualify the methodology as "gradual", it is quite different from other gradual checking work, which focuses on mixing static and dynamic checking [26]. Gradual typing has been extended to a whole range of rich type disciplines: typestates [12, 30], information flow typing and security types [8, 9], ownership types [24], annotated type systems [28], and effects [3], but not to a full-blown dependently-typed language.

This work is directly related to the work of Ou et al. on combining dependent types and simple types [18], as well as the work on hybrid type checking [17], as supported in Sage [15]. Ou et al. develop 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, τ) to check at runtime that e produces a value of (possibly-dependent) type τ. The semantics of the source language is given by translation to an internal language, which uses a type coercion judgment that inserts runtime checks when needed. In hybrid type checking, the language includes arbitrary refinements on base types, and the type system tries to statically decide implications between predicates using an external theorem prover. If it is not statically possible to either verify or refute an implication, a cast is inserted to defer checking to runtime.

In both approaches, refinements are directly expressed in the base language, as boolean expressions; therefore it suffices to evaluate the refinement expression itself at runtime to dynamically determine whether the refinement holds. (In hybrid type checking, refinements are not guaranteed to terminate, while in Ou et al., refinements are drawn from a pure subset of expressions.) In both cases, arbitrary logical properties cannot be expressed: the refinements directly correspond to boolean decision procedures, without the possibility to specify their logical meaning (see also Appendix A for a discussion of boolean reflection). In particular, there are no ways for programmers to give proof terms explicitly, which means that it is impossible to marry non-decidable (explicitly proven) properties with decidable ones (which may voluntarily be proven or deferred).

12. Conclusion

We exposed an approach to support gradual certified programming in Coq. When initially engaging in this project, we anticipated a painful extension to the theory and implementation of Coq. Much to our surprise, it was possible to achieve our objectives in a simple and elegant (albeit slightly heretical) manner, exploiting axioms and type classes. The cast framework is barely over 50 lines of Coq, to which we have to add the expansion of the Coq/HoTT Decidable library, which is useful beyond this work, and could be replaced by a different decidability framework. A limitation of the internalized approach is that it does not support blame assignment [10], because it would be necessary to modify reduction to track blame labels transparently.

An interesting track to explore is to make the axiomatic approach to casts less heretical, by requiring the claimed property to be inhabited (this would rule out direct claims of False, for instance). The counterpart is that it requires some additional effort from the programmer—it may be possible to automatically find witnesses in certain cases. Also, the monadic version seems perfectly reasonable if extraction is the main objective, because upon extraction we can eliminate the success case of the error monad, and turn the failure case into a runtime exception. Additionally, the decidability constraint could be relaxed by only requiring a sound approximation of the property to be decidable, not necessarily the property itself. Finally, we can optimize the cast procedure so that it does not execute the decision procedure if the property has been statically proven.

Acknowledgments

We thank Jonathan Aldrich, Rémi Douence, Stéphane Glondu, Ronald Garcia, François Pottier, Ilya Sergey and Matthieu Sozeau for providing helpful feedback on this work and article. Ilya Sergey also integrated the cast framework with Ssreflect as a decidability framework. We also thank the anonymous DLS reviewers, and the participants of the Coq workshop 2015 participants for their feedback, especially Georges Gonthier and Gabriel Scherer who suggested very interesting venues for future work.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, Apr. 1991.

[2] E. Allende, J. Fabry, R. Garcia, and É. Tanter. Confined gradual typing. In *Proceedings of the 29th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2014)*, pages 251–270, Portland, OR, USA, Oct. 2014. ACM Press.

[3] 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.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2):8:1–8:45, Jan. 2011.

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

[6] R. Chugh, P. M. Rondon, A. Bakst, and R. Jhala. Nested refinements: a logic for duck typing. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2012)*, pages 231–244, Philadelphia, USA, Jan. 2012. ACM Press.

[7] M. Dénès, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. QuickChick: Property-based testing for Coq. In *Coq Workshop*, 2014.

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

[9] 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.

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

[11] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI '91)*, pages 268–277. ACM Press, 1991.

[12] R. Garcia, É. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12:1–12:44, Oct. 2014.

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

[14] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming*, 23(4):357–401, 2013.

[15] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[16] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

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

[18] 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.

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

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

[21] A. Saïbi. Typing algorithm in type theory with inheritance. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL 97)*, pages 292–301, Paris, France, Jan. 1997. ACM Press.

[22] T. Schrijvers and B. C. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming (ICFP 2011)*, pages 32–44, Tokyo, Japan, Sept. 2011. ACM Press.

[23] E. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In J. Vitek, editor, *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015)*, volume 9032 of *Lecture Notes in Computer Science*, London, UK, Mar. 2015. Springer-Verlag.

[24] I. Sergey and D. Clarke. Gradual ownership types. In H. Seidl, editor, *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 579–599, Tallinn, Estonia, 2012. Springer-Verlag.

[25] T. Sheard, A. Stump, and S. Weirich. Language-based verification will change the world. In *Proceedings of the FSE/SDP Workshop on the Future of Sofware Engineering Research (FoSER 2010)*, pages 343–348, 2010.

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

[27] 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.

[28] P. Thiemann and L. Fennell. Gradual typing for annotated type systems. In Z. Shao, editor, *Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP 2014)*, volume 8410 of *Lecture Notes in Computer Science*, pages 47–66, Grenoble, France, 2014. Springer-Verlag.

[29] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming Languages and Systems (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228, Rome, Italy, Mar. 2013. Springer-Verlag.

[30] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In M. Mezini, editor, *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483, Lancaster, UK, July 2011. Springer-Verlag.

[31] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI '98)*, pages 249–257. ACM Press, 1998.

A. A Note on Boolean Reflection

An alternative approach for the definition of decision procedures is to use *boolean reflection*, i.e. considering that the decision procedure *is* the property.

```
Instance Decidable_bool (b : bool) :
    Decidable (if b then True else False).
```

However, while using boolean reflection can be convenient, there is no "safeguard" that the procedure is correctly implemented: the implementation *is* the specification. Another limitation is that the information reported to the programmer is unhelpful: if the cast succeeds, the proof term is *I*; if the cast fails, the failed property is False. While the proof term is arguably irrelevant, the information about the failed property can be very helpful for debugging.

Both issues can nevertheless been solved by having both the boolean and the property, and formally establishing the relation between both, similarly to what is done in the Ssreflect [13] library or the reflect inductive in Coq. This boolean/proposition relation mechanism is also provided in the DecidableClass library of Coq. To avoid name conflicts (the class is also named Decidable), we provide the same class under the name Decidable_relate:

```
Class Decidable_relate (P : Prop) := {
   Decidable_witness: bool;
   Decidable_spec: Decidable_witness = true ↔ P
}.
```

Actually the two presentations of decidability are equivalent. Indeed, the same development has been done in Ssreflect[17], using canonical structures [21] instead of type classes to automatically infer complex decision procedures from simpler ones [14]. This shows that the decidability mechanism is orthogonal to the cast operators we propose. [18]

B. A Note on Evaluation Regimes

Recall that Coq does not impose any fixed reduction strategy. Instead, Eval is parameterized by a reduction strategy, called a conversion tactic, such as `cbv` (aka. `compute`), `lazy`, `hnf`, `simpl`, etc.

In addition to understanding the impact of reduction strategies on the results of computations with casts, it is crucial to understand the impact of representing cast failures through an axiom. Consider a function g that expects a $\{n{:}\mathsf{nat} \mid n > 0\}$, but actually never uses its argument:

```
Definition g (x: {n:nat | n > 0}) := 1.
```

Typically, one would expect that evaluating g (? 0) with a `lazy` reduction would produce 1, while using an eager strategy like `compute` would reveal the failed cast. However:

```
Eval compute in g (? 0).
```

```
= 1
: nat
```

The reason is that a cast error in Coq is not an error per se (Coq has no such mechanism): it is just a non-canonical normal form. Therefore, even with an eager strategy, g (? 0) simply returns 1. The cast is eagerly evaluated, and fails; but this only means that g is called with `failed_cast` as a fully-evaluated argument. Because g does not touch its argument, the cast failure goes unnoticed.

On the contrary, if we extract the code to Ocaml (recall Section 8), the cast violation is reported immediately as an exception:

```
Definition client (x: nat) := g (? x).
Extraction Language Ocaml.
Extraction "test.ml" client.
```

```
# client 1;;
- : int = 1
# client 0;;
Exception: Failure "Cast has failed".
```

While, as expected, the error goes unnoticed in Haskell, because of its lazy evaluation regime.

```
Extraction Language Haskell.
Extraction "test.hs" client.
```

```
*Test> client 1
1
*Test> client 0
1
```

Extraction of axioms in eager languages. There is one last detail to discuss when considering extraction to eager languages. As defined, *failed_cast* and *cast* are extracted as follows in Ocaml:

```
let failed_cast =
  failwith "Cast has failed"

let cast dec a =
  match dec a with
  | Inl _ -> a
  | Inr _ -> failed_cast
```

While these definitions are perfectly fine for a lazy language like Haskell, in an eager language like Ocaml or Scheme they imply that loading the definition of `failed_cast` fails directly. The solution is to enforce the inlining of *failed_cast*:

```
Extraction Inline failed_cast.
```

As a result, *failed_cast* is not extracted as a separate definition, and *cast* uses the Ocaml `failwith` function directly.

[17] The Ssreflect implementation was done by Ilya Sergey.

[18] The Decidable library is currently much less furnished than the Ssreflect library using boolean reflection, but its extension with instances similar to the ones implemented in Ssreflect is straightforward.