Abstract

"Language-based verification will change the world." In their FOSER 2010 vision paper, Sheard, Stump and Weirich argue that language-based approaches to verification are bound to succeed in mainstream use. We add: provided that a gradual path to verification is supported. We explore gradual certified programming in Coq, supporting two main use cases. First, programmers can develop certified components in Coq, extract them, and ensure that the assumptions made by these components will be checked dynamically when interacting with non-certified components. Second, within Coq, we provide the possibility to postpone the proofs of selected properties, and check at “runtime” whether the properties actually hold. 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.

1. Introduction

In Certified Programming with Dependent Types, Chipala sketches two main approaches to certified programming [2]. One can write a program, its specification, and then prove separately that the program meets its specification. Another approach is to mix programs with precise specifications as part of their types, much as advocated by Sheard et al. [7]. While rich types are certainly 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, just like gradual typing allows for a smooth transition from dynamically-typed to statically-typed programs [8]. 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 [4], information flow typing [3], ownership types [6], and effects [1].

In this work, we consider a gradual path to certified programming in Coq, with two main use cases in mind. First, programmers should be able to exploit program extraction to integrate certified components within a standard system developed in a mainstream programming language, while ensuring that the assumptions made by the certified components will be checked dynamically if needed. Second, within Coq itself, programmers should be able to safely postpone having to provide some proof terms. Neither of these scenarios are however supported in Coq currently.

To illustrate these problems and our solution, 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 \(P a\). We denote the first and second projections as \(\cdot_1\) and \(\cdot_2\) respectively.

The limitations of Coq regarding a gradual approach to certified programming manifest as follows:

1. Suppose a function divide of type \(\text{nat} \rightarrow \{n: \text{nat} \mid n > 0\} \rightarrow \text{nat}\) 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 to a mainstream programming language like Ocaml or Haskell. Ideally, we would like to extract divide to a Haskell function that checks at runtime that the second argument is indeed positive, or fails otherwise:

```haskell
divide m n | n > 0 = ...
| otherwise = error "n is not > 0"
```

However, program extraction in Coq does not introduce any such safeguard. This is because of the distinction that Coq establishes between programs (in \(\text{Type}\)), which have computational content, and proofs (in \(\text{Prop}\)), which are devoid of computational meaning and are therefore erased during extraction. But in the case of decidable properties, the assumptions made by certified components should possibly be enforced in the extracted code.

2. Within Coq, providing a value of a subset 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 \texttt{Program}, a facility that allows automatic coercions to subset types leaving proof obligations to be fulfilled after the definition is completed [9]. 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 use \texttt{admit}, 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 \texttt{unsafe}: there is no delayed checking of the unproven property. For instance, one can lie about the fact that 0 is strictly positive using \texttt{admit}, and may apply divide without ever noticing the violation, possibly getting an incorrect result. The motto of gradual checking, \texttt{trust but verify}, is therefore not supported. Note that because Coq is dependent-type, there is no rigid compile-time/runtime distinction: therefore, such validation errors could possibly occur both as part of standard evaluation (triggered with \texttt{Eval}) and as part of type checking, during type conversion.

2. An Axiomatic Approach to Casts

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

\(^1\)Note that we use the name “cast” in the standard way [5], 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.
this is feasible entirely within standard Coq, without extending the underlying theory and implementation.

1. To address the first point above, we can cast `divide` from type `nat → {n:nat | n > 0} → nat` to the plain type `nat → nat → nat` in a safe manner: when applied, the casted function checks that its argument is strictly positive, and raises a runtime cast error otherwise.

2. In Coq, when casting a value `a` of type `A` to the rich type `{a:A | P a}`, the property `P a` is checked as needed, forbidding unsafe projection of the value of type `A` from the dependent pair.

An heretical approach. A controversial key feature of our development is that we support a smooth gradual path to certified programming by avoiding the monadic style of programming to handle cast errors. Instead, we represent cast failure in Coq as an inconsistent axiom, so that failed casts manifest as non-canonical normal forms in Coq, and as runtime exceptions in the extracted code.

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 | P a}`:

\[
\text{Axiom failed_cast:} \quad \forall \{A:\text{Type}\} \{P : A \rightarrow \text{Prop}\} \{a:A | P a\}.
\]

This axiom is obviously a lie! But this lie allows us to provide a cast operator as a function of type `A → {a:A | P a}`, even though a cast can fail. This means programmers can use cast without having to deal with optional values as a monadic approach would entail.

\[
\text{Definition cast (A:Type) (P : A \rightarrow \text{Prop}) (dec : \forall a, \text{Decidable (P a)}) : A \rightarrow \{a:A | P a\} :=}
\]

\[
\text{fun a:A →}
\]

\[
\text{match dec a with}
\]

\[
\text{| inl p ⇒ (a : p) }
\]

\[
\text{| inr_ ⇒ failed_cast (A a) (P a)}
\]

\[
\text{end.}
\]

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. (We extend the Coq/HoTT Decidable type class library.)

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 gradual verification framework in the context of program extraction. Indeed, with extraction, all properties (in `Prop`) are erased; so a casted value would not be extracted to a value associated with a runtime check, but just to a plain, unchecked value. Even within Coq, 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).

Advanced cast operators. Building on top of the `cast` operator, we define higher-order operators for casting function types, including dependently-typed functions. These operators include both strengthening the range of a function (ie. claiming that the function returns values of richer types) and widening its domain (ie. allowing a function that accepts arguments of rich types to accept arguments of plain types).

The domain widening cast is particularly helpful for program extraction, as discussed above. Casting `divide` from type `nat → {n:nat | n > 0} → nat` to the plain type `nat → nat → nat` results in a function wrapper that, when extracted, is a function that dynamically checks that the second argument is strictly positive, or raises a runtime cast error otherwise.

We can also extend our approach to deal with other dependently-typed constructions beyond subset types, such as records.

Implementation. We have implemented a Coq library for gradual certified programming, called Cocasse. The cast framework itself is barely over 50 lines of Coq, to which we have to add the expansion of the Coq/HoTT Decidable library, which could be replaced by a different decidability framework. Cocasse is available at `https://github.com/tabareau/Cocasse`. The distribution includes several examples of use of the different cast operators, including dependent functions and records.

Benefits of being heretical. The axiomatic approach to casts trades consistency for flexibility. The alternative is a monadic approach, in which cast operators like `cast` return an optional value. 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 extract the value component of a subset type with a failed cast (using `admit` provides no such guarantee).
- When things go right (i.e. the cast succeeds), there is no axiom or admitted definition that will block type conversion and evaluation.
- Through program extraction, the use of admitted properties is simply and unsafely erased. Conversely, because casts establish a bridge between properties and computation, they are extracted as runtime checks, and cast failures manifest as runtime exceptions—which is exactly how standard casts work in mainstream programming languages.

All in all, both the monadic and axiomatic approaches to gradual verification are feasible, and are likely to please different crowds. We focus on the axiomatic approach, because of its disruptive potential and software engineering benefits. We believe this approach will be appealing to pragmatic practitioners who are willing to compromise consistency to some extent in order to enjoy a smooth gradual verification environment.

References