

Gradual Certified Programming in Coq

joint work with Nicolas Tabareau (INRIA)

1

Coq

interactive theorem prover with some support for automation

pure functional programming language

most expressive type system around (CIC)

- type checker ok \Rightarrow program “correct”

2

Certified Programming

Develop programs in Coq with rich semantic properties

1. write programs, specs, prove properties
2. mix

e-voting???

Extract them to practical and efficient languages like OCaml and Haskell

3

Gradual Certified Programming

Certified programming is great and promising

- but quite challenging!

Support a gradual path to expressive properties

- “typed \rightarrow very typed” (not interested in untyped)

4

Refinements in Coq

dependent pairs

- sigma types: $\Sigma t:T. P(t)$
- in Coq: written $\{t : T \mid P\ t\}$, aka. “subset types”
- inhabitant: $(x ; p)$ where p is a proof of $P(x)$

5

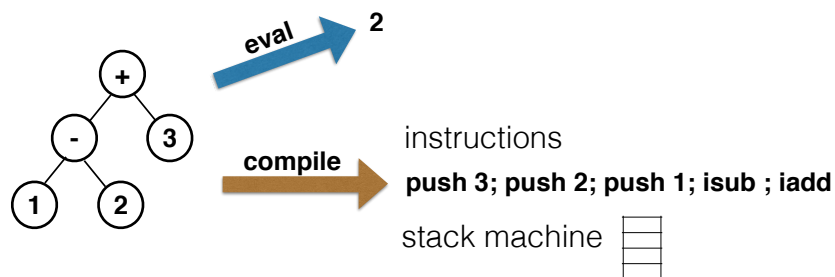
Motivation

Write rich types without necessarily providing all the proofs

- not like plain admit: verify later, when/if needed
- allows to use testing to get evidence for the (in)correctness of the stated property

6

a certified compiler for a small language



7

Motivation

Integration of certified components with plain ones

- extract certified components to Ocaml/Haskell/Scheme
- build a whole system by combining components
- protect assumptions of certified components from misuse


8

Illustration

dependently-typed stack machine

instructions

push 3; push 2; push 1; isub ; iadd

stack machine 

9

Problem

Properties and type dependencies are lost upon extraction

Need a way to “protect” extracted components

(note: crash happened with pure code!)

10

Example

`divide : nat → {n:nat | n > 0} → nat`



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

11

Key ideas

Turn properties into runtime checks

Hide the potential for cast errors to support smooth integration

12

Decidability

dynamic check only makes sense if P is decidable

- if so, then evaluate its decision procedure

Class `Decidable` ($P : \text{Prop}$) := `dec` : $P + \neg P$.

derive complex decision procedures automatically

Instance `Decidable_and` ($P Q : \text{Prop}$) ($HP : \text{Decidable } P$)
 ($HQ : \text{Decidable } Q$) : `Decidable` ($P \wedge Q$).

13

Casts

How to represent the potential for errors?

- error monad
- cast: $A \rightarrow \text{option } \{a:A \mid P a\}$
- changes the interface of components

Seamless, but heretical alternative: pose an axiom!

- cast: $A \rightarrow \{a:A \mid P a\}$

14

Casts

Axiom `failed_cast` :

$\forall \{A:\text{Type}\} \{P : A \rightarrow \text{Prop}\} (a:A) (msg:\text{Prop}), \{a:A \mid P a\}.$

Definition `cast` ($A:\text{Type}$) ($P : A \rightarrow \text{Prop}$)

($dec : \forall a, \text{Decidable } (P a)$) : $A \rightarrow \{a:A \mid P a\} :=$

`fun` $a:A \Rightarrow$

`match` $dec\ a$ `with`

| `inl` $p \Rightarrow (a ; p)$

| `inr` $_ \Rightarrow \text{failed_cast } a (P a)$

`end.`

15

Higher-order Casts, simple

Definition `cast_fun_range` ($A B : \text{Type}$) ($P : B \rightarrow \text{Prop}$)

($dec : \forall b, \text{Decidable } (P b)$) :

$(A \rightarrow B) \rightarrow A \rightarrow \{b:B \mid P b\} :=$

`fun` $f\ a \Rightarrow ? (f\ a).$

Definition `cast_fun_dom` ($A B : \text{Type}$) ($P : A \rightarrow \text{Prop}$)

($dec : \forall a, \text{Decidable } (P a)$) :

$(\{a:A \mid P a\} \rightarrow B) \rightarrow A \rightarrow B :=$

`fun` $f\ a \Rightarrow f\ (? a).$

16

Higher-order Casts, dependent

Widening the domain of dependently-typed functions is more tricky

- the “lie” about casts percolates at the type level!

```
Definition cast_forall_dom (A: Type) (P: A → Prop)
  (B: A → Type) (dec: ∀ a, Decidable (P a)) :
  (∀ x: {a: A | P a}, B x.1) → (∀ a: A, B a) :=
  fun f a => f (? a).
```

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

(need a second axiom, which cannot fail in an eager language)

17

Implicit casts

Gradual typing typically implies *implicit* cast insertion

- Coq has implicit coercions
- can use them to mimic a more transparent gradual system

```
Definition nat_to_rnat : nat → rich_nat := ?.
Coercion nat_to_rnat : nat ↦ rich_nat.
Variable g : rich_nat → nat.
Variable n : nat.
Check g n.
```

18

Properties

- canonicity of Coq: the only non-canonical terms come from the use of axioms
- within Coq, a cast failure is a use of an axiom:
t = E[failed_cast(p)]
- through extraction: gives the usual gradual theorem (only errors are cast errors, safe otherwise)
- termination of casts (unlike hybrid typing in Sage)
- interaction with other components: can be broken through mutation...

19

Perspectives

Can extend this to rich records (eg. algebraic structures)

How to detect the use of lies (axioms)?

How to deal with arbitrary type dependencies?

Can we protect certified components from arbitrary imperative code?

20