Thoughts on Programming with Proof Assistants

Adam Chlipala
University of California, Berkeley

PLPV Workshop

Not Ready for Prime Time?

Proof assistants like Coq are useful for doing math, but they're far too inconvenient for serious dependently typed programming!

Just using “refinement

general.

Need to span many levels of abstraction, so good modularization is key to feasibility

Or are they?

Cons

- No imperativity, general recursion, or exceptions
- Very primitive dependent pattern matching

Pros

- Easy to combine programming with tactic-based proving
- A mature set of tools for proof organization and automation

This Talk: Capsule summary of my experiences implementing Proof-Carrying Code-style program verifiers in Coq using dependent types to guarantee total correctness.

Mixing Programming with Tactics

```
Definition isEven : forall n, [even(n)].
  refine (fix isEven (n : nat)
      : [even(n)] =
    match n retur     even(n)
      O -> Yes
    | S O ->
    | S (S n
      proof <- I
      Yes);
  aut
Qed.
```

The type of an optional proof of a proposition

Step 1. Declare the func a

Step 3. Generat *roof part"* of the actics.

Generate a proof obligation

Result: A term in the Calculus of Inductive Constructions

Missing?

Imperativity?

Pure functional data structures worked well for all of the situations I encountered.

Non-termination and general recursion?

The kinds of program analysis algorithms I needed were naturally primitive recursive.

Exceptions?

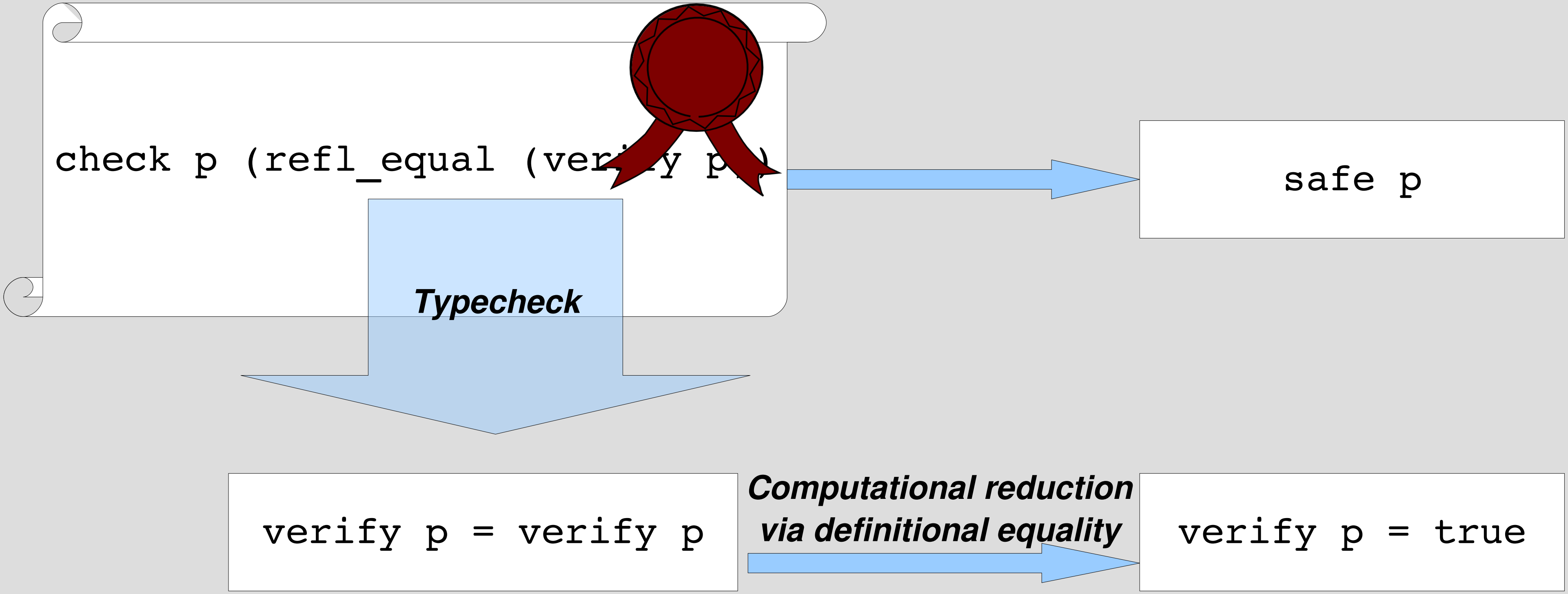
Failure monads provide a cleaner alternative to “exceptional” uses of exceptions.

Fancy dependent pattern matching?

Sticking to refinement types, vanilla pattern matching is good enough.

Reflective Proofs

```
Theorem check : forall (p : program),
    verify p = true
    -> safe p.
```



```
check p (refl_equal (verify p))
```

```
safe p
```

Typecheck

```
verify p = verify p
```

Computational reduction via definitional equality

```
verify p = true
```

Other Benefits

- Module system
- Lots of pre-written proof-generating decision procedures
- Expressive tactical language
- Extensible goal-directed proof search mechanism
- Extraction to OCaml (and from there to fast native code)

Conclusion

- Consider using Coq for your next dependently typed program if large non-syntax-directed proofs are a large part of it.
- It seems worthwhile to keep in mind potential overlaps between programming environments and proof assistants in developing new PLPV tools.

For more info: See my talk at ICFP next month!