An Environment for Programming with Dependent Types, Take II

Matthieu Sozeau

Saarland University
Saarbrücken, Germany
April 27th 2018

Inria Paris & IRIF, Université Paris 7 Diderot
OUTLINE

1. Template Coq - Reification and Reflection
2. CertiCoq - Certified Compilation
3. Equations - Function Definitions and Reasoning
4. UniCoq - Unification
5. CoqHoTT - Definitional Proof-Irrelevance
6. Back to the Future
• **Reifies** and **reflects** the syntax and typing/operational semantics of Coq

• ...or rather pCuIc (Timany & Sozeau, FSCD’18): Predicative, Universe Polymorphic Calculus of Cumulative Inductive Constructions (phew…)

• Initially developed by G. Malecha

• Quoting and unquoting of terms and declarations
  
  ○ Quote Definition `quoted_t : Ast.t := t.`
  
  ○ Make Definition `denoted_t := quoted_t.`

**Typed Template Coq**

Anand, Boulier, Cohen, Sozeau & Tabareau - ITP’18
Ideally “faithful” representation of Coq terms
Differences: Strings for global_reference and lists instead of arrays. But see native integers and arrays (M. Dénès)…

Ast.v (term) &
template-demo.v
To prove interesting theorems, we also need a specification of **typing & reduction** in pCuIC.

**Current focus:** specification of pCuIC as implemented in Coq:

- Inductive specifications of typing, conversion and reduction
- Strict positivity and guard condition (w/ C. Mangin).
- **No modules yet:** PMP, Derek Dreyer, Joshua Yanovski and I have a “plan” (involving $\omega$-universes…)
- Conversion: Krivine Machine vs Parallel Reduction specification
- A (partial) type-checker written in Coq!

**Typing.v & Checker.v**
The TemplateCoq Monad

- Similar to Mtac’s monad (shallow vs deep terms)
- Allows crawling the environment and modifying it, calling the type checker etc…
- WIP OCaml version on its extraction for building plugins entirely in Coq
- Example plugins: variants of parametricity translations (Boulier, Cohen, Morrisett and Anand), forcing translation (Danil Annenkov, Nantes), ETT to ITT (Winterhalter, Tabareau and I)
Template Coq WIP

- Justify Mtac 2 programs and run them without oracles, on bare metal (with CertiCoq)?
- First need to formalize the unification algorithm (Ziliani & Sozeau, JFP’17) to actually build interesting tactics (part of CSEC program - partnership with Santiago and Nantes)
OUTLINE

1. Template Coq - Reification and Reflection
2. CertiCoq - Certified Compilation
   a. The compiler
   b. Erasure
3. Equations - Function Definitions and Reasoning
4. UniCoq - Unification
5. CoqHoTT & Definitional Proof-Irrelevance
6. Back to the Future
Gallina $\rightarrow$ Clight

\[
\text{compile} : \text{Template.Ast.term} \rightarrow \text{Compcert.Csyntax}
\]

Theorem (forward simulation): $\forall \ t \ v : \text{Ast.term},$
\[
closed t \rightarrow \\
t \sim>_\text{wcbv} v \rightarrow \\
\exists v', \text{compile } t \sim>_C v' \land v \sim v'
\]

Erases proofs, type labels, types, parameters of constructors, and lambdas of match branches, then CPS, closure conversion, shrink reduction… binding to a GC.
Extraction and Erasure

1. Extract to ML, compile and bind it to CompCert
2. ML Reifier from Coq’s constr to Template Coq’s extracted Coq_term (trivial, part of Template Coq, reusable for plugins)

Extraction in the TCB currently □
Towards Certified Extraction with Letouzey, Anand...

Bootstraping à la CakeML in the future!
Extraction and Erasure: bootstrapping

1. Run `in Coq `compile (reified_compile)`
2. # certicoqc Typechecker.v
CertiCoq Results

1. **CPS** switching to a named representation, using a template-coq plugin for parametricity by Anand and Morrisett!
   
   Open problem: do it with types.

2. **Safe-for-space** Closure-conversion & shrink reduction

3. Defunctionalization, **representation optimization** & translation to Clight.

4. Complete proof of observation preservation for **closed** inductive values and functions (for linking).

5. **WIP** linking to the VST C program logic (Z. Paraskevopoulou)
OUTLINE

1. Template Coq - Reification and Reflection
2. CertiCoq - Certified Compilation
3. Equations - Function Definitions and Reasoning
   a. Focus on Recursion
   b. Equations + CertiCoq
4. UniCoq - Unification
5. CoqHoTT & Definitional Proof-Irrelevance
6. Back to the Future
• Dependent Pattern-Matching à la Epigram, Agda
• Compiled-down to CIC using telescope simplification (à la Cockx circa 2016)
• Optional typeclass instances of K/decidable equality
• Smart case compilation: small proof terms, avoiding UIP
• Structural, nested and well-founded recursion (i.e. more than what Function/Program can handle)
• Derive Signature NoConfusion Subterm EqDec for I
• Generates graph, unfolding lemma, elimination principles
Equations and Recursion

1. **Focus on Recursion**

2. Equations + CertiCoq: towards optimizing compilation
Focus on Recursion: Nested, Mutual

http://mattam82.github.io/Coq-Equations/examples/nested_mut_rec.html

Functional elimination is good for you!
Focus on Recursion: Well-founded

structurally recursive

\[ \subseteq \]

well-founded on subterm relation

1) Derive Subterm for I relation on (computational/hType) inductive families

2) Prove well-foundedness by structural rec

3) Profit! “by rec I_subterm x”

- Define split on vectors by rec on the vector or the index!

- Extracts to general fixpoints
The Beauty of Logic

Equations elements $r : \text{rose}$ $(acc : \text{list } A) : \text{list } A :=$
elements $r \ l \ \text{by \ rec} \ r \ (\text{MR \ lt \ size}) :=$
elements $(\text{leaf } a) \ acc := a : \ : \ acc;$
elements $(\text{node } l) \ acc := aux \ l \ _$
    where aux $x \ (H : \text{list\_size \ size } x < \text{size } (\text{node } l)) : \text{list } A :=$
aux $x \ H \ \text{by \ rec} \ x \ (\text{MR \ lt \ (list\_size \ size)}) :=$
aux nil _ := acc;
aux (\text{cons } x \ xs) H := \text{elements } x \ (\text{aux } xs \ (\text{list\_size\_smaller } x \ xs \ l \ H)).$

Definition elems $r := \text{elements } r \ \text{nil}.$

- Use the weapon of your choice
- Equations generates unfolding lemma
- Eliminator abstracts away from the w.f. relation
Computational content

• Closed calls still reduce to the same normal forms: 
  \( \text{I}_{\text{subterm}} \) is closed

• Make it fast by adding \( 2^n \text{Acc}_{\text{intro}} \)'s to the well-foundedness proof.

• For calls on open terms:
  – Proofs: unfolding lemma and derived equalities
  – Programs: still reduces, unfolding might be unwieldy

• Functional extensionality is used to prove the unfolding lemma (easier to automate)
Playtime: Regexp matching

- Implement regexp matching using continuations instead of derivatives or automata (Harper’99 - “Proof-directed debugging”)
- Needs dependent types, well-founded recursion, and eliminator for recursive calls “under binders”...

Yesterday’s Tutorial
An environment for Programming with Dependent Types, Take II

type 'alpha regexp =
  | Empty
  | Epsilon
  | Atom of 'alpha
  | Disj of bool * bool * 'alpha regexp * 'alpha regexp
  | Conj of bool * bool * 'alpha regexp * 'alpha regexp
  | Seq of bool * bool * 'alpha regexp * 'alpha regexp
  | Star of 'alpha regexp

(type 'alpha substring = 'alpha list)
(type 'alpha cont_type = 'alpha substring -> bool)

(* val matches :
   'a1 alphabet -> bool -> 'a1 regexp -> 'a1 list -> 'a1 cont_type -> bool **)

let matches alpha null r s k =
let hypspock = { pr1 = null; pr2 = { pr1 = r; pr2 = { pr1 = s; pr2 = { pr1 = k; pr2 = Tt } } } } in
let rec fix_F x =
  let h = x,pr2 in
  let r0 = h,pr1 in
  let h0 = h,pr2 in
  let s0 = h0,pr1 in
  let h1 = h0,pr2 in
  let k0 = h1,pr1 in
  let matches0 = fun null0 r1 s1 k1 ->
    let y = { pr1 = null0; pr2 = { pr1 = r1; pr2 = { pr1 = s1; pr2 = { pr1 = k1; pr2 = Tt } } } } in
    (Fun _ -> fix_F y)
  in
  (Match 0 with
    | Empty -> False
    | Epsilon -> k0 s0
    | Atom l ->
      (match 00 with
        | Nil -> False
        | Cons (c, l0) ->
          (match equival_dec (alphabet_dec alpha) l a with
            | Left -> k0 l0
            | Right -> False))
        | Disj (l, r1, r2, r3) ->
          (match matches0 l r2 s0 k0 __ with
            | True -> True
            | False -> matches0 r1 r3 s0 k0 __)
        | Conj (l, r1, r2, r3) ->
          matches0 l r2 s0 (fun s' ->
            matches0 r1 r3 s0 (fun s'' ->
              match equival_dec (list_eqdec (alphabet_dec alpha)) s' s'' with
                | Left -> k0 s'
                | Right -> False __)
                __)
        | Seq (l, r1, r2, r3) ->
          let k1 = fun s' -> matches0 r1 r3 s' k0 __ in matches0 l r2 s0 k1 __
          in
          in
          in
          (fix_F hypspock)
More examples

- Hereditary substitution for Predicative System F (Mangin & Sozeau, LFMTP’15)
  Nested recursion, well-founded multiset ordering on types.
- Ordinal measures (Castéran)
- Reflexive ring-like tactic on polynomials. WF subterm order on indexed polynomials
- Prototyping without verifying termination using functional eliminator

mattam82.github.io/Coq-Equations/examples
Equations and Recursion

1. Focus on Recursion

2. Putting it all together: Equations + CertiCoq
Equations + CertiCoq

High-level dependent pattern-matching

⇒

Assembly

Goal: do better than extraction.

- Erases proofs: stuff in Prop, all equality manipulations and well-foundedness proofs (as in regular Extraction)
- Erases types (abstraction annotations, parameters)
- Representation optimization, unboxing:

  Inductive bigint :=
  | bignat (i : int63)
  | bigbig (i : BigInt.t).
Indices do not matter

But does not erase indices!

\[
\text{Inductive fin : nat } \rightarrow \text{ Type :=}
| \text{ fz (n : nat) } \\
| \text{ fs (n : nat) (f : fin n).}
\]

• **If** none of the functions on \text{fin} use the index, it is just used for typing / justifying recursion arguments.

• **Ideally should extract to**…

\[
\text{Inductive fin : Type :=}
| \text{ fz } | \text{ fs (f : fin).}
\]

• **Might require moving to** \text{CIC}^* (with a different intersection product \(\forall\)) or a modal (weighted) DTT
Indices do not matter

Dependent-types ensure our programs never go wrong, and do the right thing, statically.

⇒ Get rid of dependencies & get the (safe) code to run at full speed.

head x = match x with
  | nil ⇒ assert false
  | cons x _ ⇒ x
end

⇒

fn head (x : list) { return (*x).hd; }
Equations Summary

- Write **just what’s needed** when programming with dependently-typed structures.
- Gives the **right reasoning principles** on your (mutual, nested, dependent) function.
- CertiCoq compiles it maintaining **certification assurance**.
- Future: **run faster** that simply-typed program + correctness proof.
- Good **target** for verification of total, purely functional **Haskell** programs (e.g. hs-to-coq, UPenn).
OUTLINE

1. Template Coq - Reification and Reflection
2. CertiCoq - Certified Compilation
3. Equations - Function Definitions and Reasoning
4. UniCoq - Unification
5. CoqHoTT & Definitional Proof-Irrelevance
6. Back to the Future
UniCoq
j.w.w. Beta Ziliani

A paper formalisation and re-implementation of the unification algorithm of Coq.

- Higher-Order Unification
- Pruning and Dependency Erasure Heuristic
- First-Order Approximation
- Universes

Used in Mtac 2
**WIP**: formalize it on top of Template Coq

- **Verify** metatheoretical properties
- **Certified implementation:**
  A formal specification for a highly sensitive part of the system: proof developer/software interface.
- A tool to develop higher-level plugins: Mtac or Beluga embeddings, **DSPLs**: Domain-Specific Proof Languages.
OUTLINE

1. Template Coq - Reification and Reflection
2. CertiCoq - Certified Compilation
3. Equations - Function Definitions and Reasoning
4. UniCoq - Unification
5. CoqHoTT & Definitional Proof-Irrelevance
6. Back to the Future
Definitional Proof-Irrelevance:

\[ \Gamma \vdash P : \text{sProp} \quad \Gamma \vdash t, u : P \]

\[ \Gamma \vdash t \equiv u : P \]

- Universe of **strict** propositions, with definitional **UIP**
- For subset types:
  \[ \forall \ p \ q : P \ x, (x, p) \equiv (x, q) : \{ x : A \mid P \} \]
- Faithful target of **Program**
Technically

Annotate binders with sort information:

\[ \Gamma, \, x \, : \, ^* \, \text{True} \, \vdash \, t \, : \, A \]

-----------------------------

\[ \Gamma \, \vdash \, \lambda \, x \, : \, ^* \, \text{True}. \, t \, : \, \Pi \, _\, _\, : \, ^* \, \text{True}, \, A \]

- Conversion ignores proofs, i.e. “Extraction” during conversion
- sProp universe extensible with **decideably invertible** propositions, e.g.

\[ \text{le} \, : \, \text{nat} \to \text{nat} \to \text{Prop} \]
Results

• More type conversions, more efficiently
• Easier to work with **coercions** (**transports** along sSet indices)
• Closer to the extracted computational behavior
• **WIP**: Homotopy-compatible model (more refined than 2-level type theory)
• **WIP**: Relation to irrelevance in Agda (j.w.w. Jesper Cockx)
OUTLINE

1. Template Coq - Reification and Reflection
2. CertiCoq - Certified Compilation
3. Equations - Function Definitions and Reasoning
4. UniCoq - Unification
5. CoqHoTT & Definitional Proof-Irrelevance
6. Back to the Future
Back to the Future (WIP)

A completely certified toolchain for DTP:

1. **Kernel**: type/proof checker
   – Extracted from Template Coq type checker
2. Certified optimizing **compiler** for efficient execution
   – Bootstrapped CertiCoq compiler
3. **Unification** and higher-level **tactics** – UniCoq in Coq
4. **Definitional translation** of definitions by dependent pattern-matching and recursion – Equations
5. A **definitional Proof-Irrelevance** extension for easier reasoning on dependently-typed programs.
# opam install
coq-template-coq
coq-equations
coq-unicoq