Towards verified extraction from Coq to OCaml

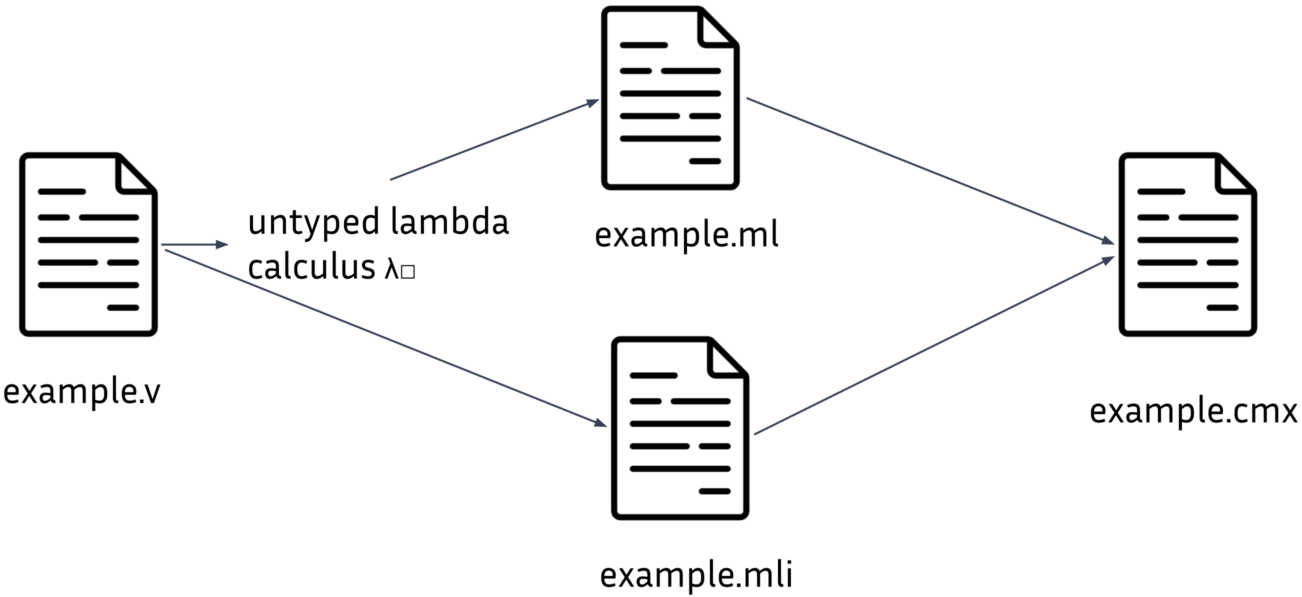
Yannick Forster, Matthieu Sozeau, Pierre-Marie Pedrót, and Nicolas Tabareau

Gallinette Team, Inria Nantes

Talk at the Cambium seminar, December 7th

received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101024493.

Extraction in Coq



Extraction in Coq

Coq's Extraction turns 18 this year!

One of the central claims to fame of Coq

N° d'ordre : 7567

Thèse de doctorat

présentée à

L'Université de Paris-Sud

U.F.R. Scientifique d'Orsay

par

PIERRE LETOUZEY

pour obtenir

le grade de docteur en sciences
de l'Université de Paris XI Orsay
spécialité : Informatique

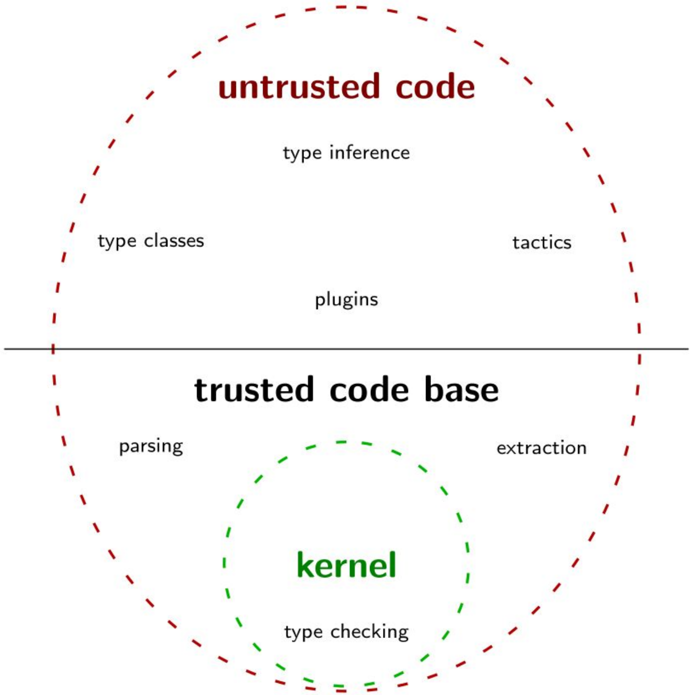
Sujet :

Programmation fonctionnelle certifiée
L'extraction de programmes dans l'assistant Coq

Soutenue le 9 juillet 2004 devant la commission d'examen composée de

M.	LEROY Xavier	président
M.	BERARDI Stefano	rapporteurs
M.	MONIN Jean-François	
Mme	BENZAKEN Véronique	examinateurs
M.	SCHWICHTENBERG Helmut	
Mme	PAULIN Christine	directeur

The ideal of proof assistants



The underlying type theory

W-Empty

$$\frac{}{\mathcal{WF}([])[]}$$

W-Local-Assum

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma :: (x : T)]}$$

W-Local-Def

$$\frac{E[\Gamma] \vdash t : T \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma :: (x := t : T)]}$$

W-Global-Assum

$$\frac{E[] \vdash T : s \qquad s \in \mathcal{S} \qquad c \notin E}{\mathcal{WF}(E;\ c : T)[]}$$

W-Global-Def

$$\frac{E[] \vdash t : T \qquad c \notin E}{\mathcal{WF}(E;\ c := t : T)[]}$$

Ax-SProp

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{SProp} : \mathsf{Type}(1)}$$

Ax-Prop

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Prop} : \mathsf{Type}(1)}$$

Ax-Set

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Set} : \mathsf{Type}(1)}$$

Ax-Type

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Type}(i) : \mathsf{Type}(i+1)}$$

Var

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x : T) \in \Gamma \text{ or } (x := t : T) \in \Gamma \text{ for some } t}{E[\Gamma] \vdash x : T}$$

Const

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c : T) \in E \text{ or } (c := t : T) \in E \text{ for some } t}{E[\Gamma] \vdash c : T}$$

Prod-SProp

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{SProp}}{E[\Gamma] \vdash \forall\, x : T, U : \mathsf{SProp}}$$

Prod-Prop

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Prop}}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Prop}}$$

Prod-Prop

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Prop}}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Prop}}$$

Prod-Set

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathsf{SProp}, \mathsf{Prop}, \mathsf{Set}\} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Set}}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Set}}$$

Prod-Type

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathsf{SProp}, \mathsf{Type}(i)\} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Type}(i)}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Type}(i)}$$

Lam

$$\frac{E[\Gamma] \vdash \forall x : T,\ U : s \qquad E[\Gamma :: (x : T)] \vdash t : U}{E[\Gamma] \vdash \lambda x : T.\ t : \forall x : T,\ U}$$

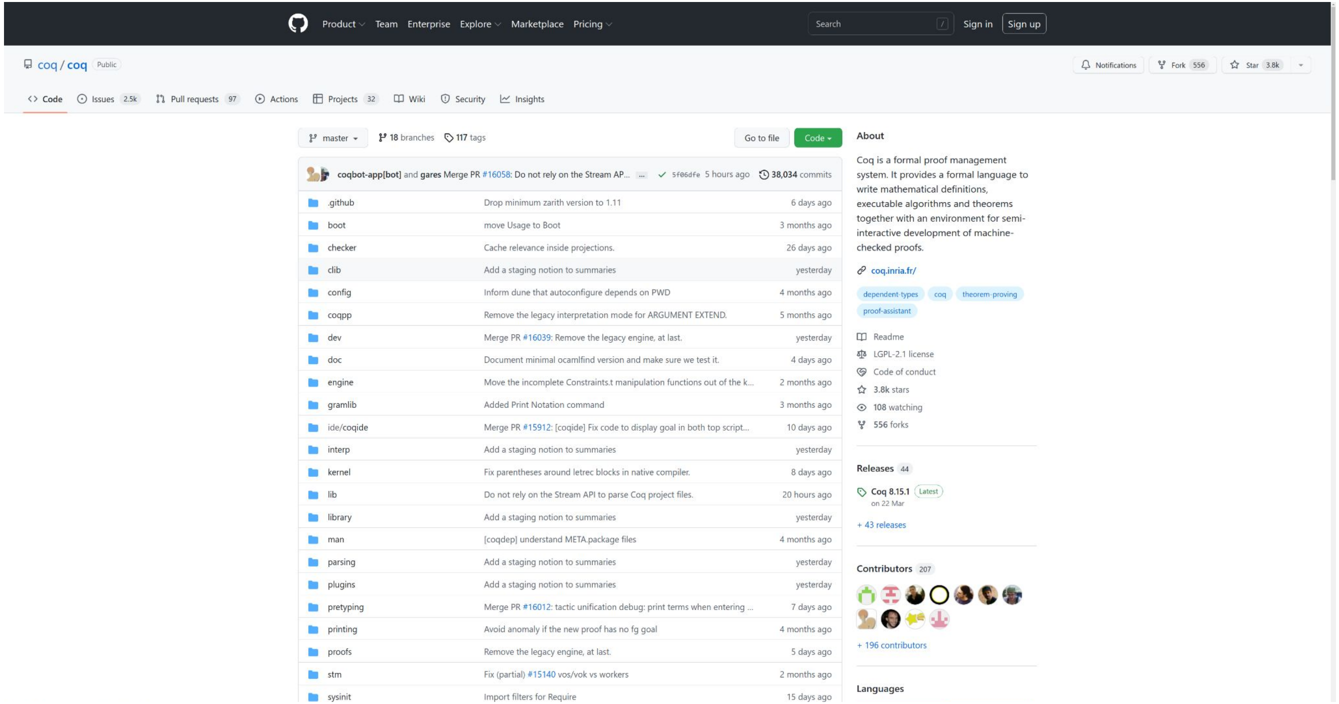
App

$$\frac{E[\Gamma] \vdash t : \forall x : U,\ T \qquad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{x/u\}}$$

Let

$$\frac{E[\Gamma] \vdash t : T \qquad E[\Gamma :: (x := t : T)] \vdash u : U}{E[\Gamma] \vdash \mathsf{let}\ x := t : T\ \mathsf{in}\ u : U\{x/t\}}$$

The implementation



About 200k LoC - about 1 critical bug per year

Issues with implementation of extraction

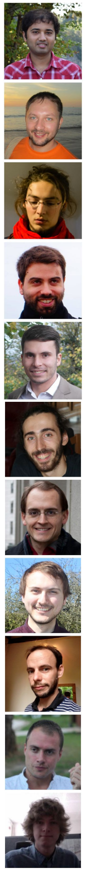
- practical: has bugs
- strategic: is unmaintained
- conceptual: inserts lots of Obj.magic
- missing features: e.g. no GADTs

The vision

Give a verified implementation of extraction

- formalise Coq in Coq
- formalise (a variant of) OCaml
- re-implement extraction
- verify it

The MetaCoq project



- a formalisation of Coq in Coq
  - confluence, validity, subject reduction
  - weak call-by-value standardisation (if t is of first-order inductive type and reduces to a value, then this value can be found with weak call-by-value evaluation)
- machine-checked programs regarding Coq:
  - a correct and complete type checker
  - an erasure procedure into an untyped version of Coq, removing proofs
- Vision: a fast kernel for daily use, a verified kernel for monthly use
- Future work:
  - eta, WIP by Meven Lennon-Bertrand
  - SProp, WIP by Yann Leray
  - modules, WIP by Yee Jian Tan
  - template polymorphism, subsumed by sort polymorphism, WIP by Kenji Maillard et al

Template-Coq

```
Inductive term : Type :=
    tRel : nat -> term
  | tVar : ident -> term
  | tEvar : nat -> list term -> term
  | tSort : Universe.t -> term
  | tCast : term -> cast_kind -> term -> term
  | tProd : aname -> term -> term -> term
  | tLambda : aname -> term -> term -> term
  | tLetIn : aname -> term -> term -> term -> term
  | tApp : term -> list term -> term
  | tConst : kername -> Instance.t -> term
  | tInd : inductive -> Instance.t -> term
  | tConstruct : inductive -> nat -> Instance.t -> term
  | tCase : case_info -> predicate term -> term ->
            list (branch term) -> term
  | tProj : projection -> term -> term
  | tFix : mfixpoint term -> nat -> term
  | tCoFix : mfixpoint term -> nat -> term.
```

Erasure to lambda box (Coq Coq Correct @ POPL 20)

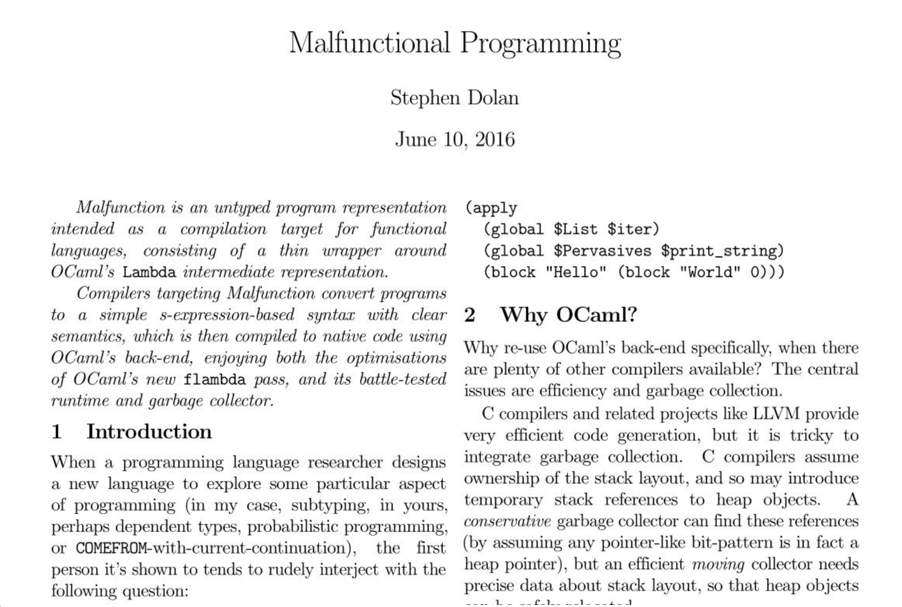
We implemented an erasure function from well-typed terms to lambda box in Coq, following Letouzey's proof.

Theorem: Let $\Sigma; \Gamma \vdash t : T$ and T be a first-order type. If t reduces to an irreducible term v, then the erasure of t weak call-by-value evaluates to the erasure of v.

The vision

Give a verified implementation of extraction

- formalise Coq in Coq
- **formalise (a variant of) OCaml**
- re-implement extraction
- verify it



Malfunctional Programming

Stephen Dolan

June 10, 2016

Malfunction is an untyped program representation intended as a compilation target for functional languages, consisting of a thin wrapper around OCaml's `Lambda` *intermediate representation.*

Compilers targeting Malfunction convert programs to a simple s-expression-based syntax with clear semantics, which is then compiled to native code using OCaml's back-end, enjoying both the optimisations of OCaml's new `flambda` *pass, and its battle-tested runtime and garbage collector.*

1 Introduction

When a programming language researcher designs a new language to explore some particular aspect of programming (in my case, subtyping, in yours, perhaps dependent types, probabilistic programming, or COMEFROM-with-current-continuation), the first person it's shown to tends to rudely interject with the following question:

```
(apply
  (global $List $iter)
  (global $Pervasives $print_string)
  (block "Hello" (block "World" 0)))
```

2 Why OCaml?

Why re-use OCaml's back-end specifically, when there are plenty of other compilers available? The central issues are efficiency and garbage collection.

C compilers and related projects like LLVM provide very efficient code generation, but it is tricky to integrate garbage collection. C compilers assume ownership of the stack layout, and so may introduce temporary stack references to heap objects. A *conservative* garbage collector can find these references (by assuming any pointer-like bit-pattern is in fact a heap pointer), but an efficient *moving* collector needs precise data about stack layout, so that heap objects

```
Type t :=
Inductive t :=
| Mvar of Ident.t
| Mlambda of Ident.t list * t
| Mapply of t * t list
| Mlet of binding list * t
| Mnum of numconst
| Mstring of string
| Mglobal of Longident.t
| Mswitch of t * (case list * t) list
(* Numbers *)
| Mnumop1 of unary_num_op * numtype * t
| Mnumop2 of binary_num_op * numtype * t * t
| Mconvert of numtype * numtype * t
(* Vectors … *)
(* Lazy … *)
(* Blocks *)
| Mblock of int * t list
| Mfield of int * t
with binding :=
and binding =
| Unnamed of t | Named of Ident.t * t | Recursive of
(Ident.t * t) list
[ `Unnamed of t | `Named of Ident.t * t |
`Recursive of (Ident.t * t) list ]
```

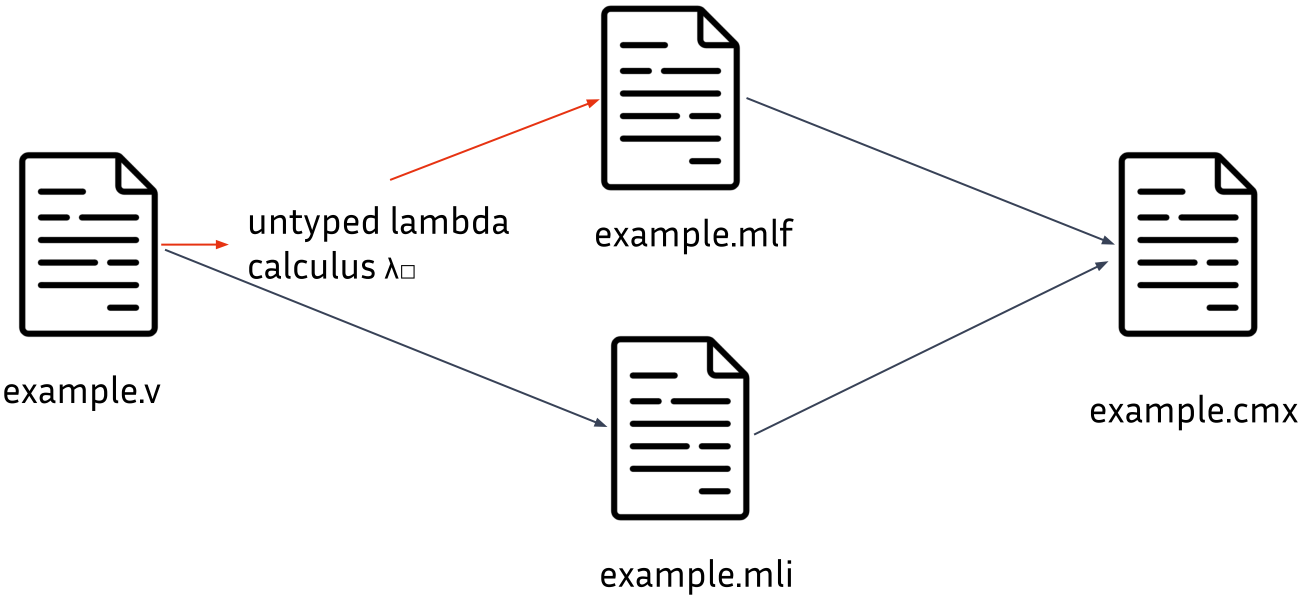
Malfunction

Specification: Interpreter using references for recursion

Plan:

1. Make the Malfunction interpreter pure (recursion via let rec), test it
2. Implement interpreter in Coq, extract and test it
3. Define inductive evaluation relation, verify it

Extraction in Coq, using MetaCoq and Malfunction



Almost^{TM} verified extraction to OCaml

Theorem: Given

- a first-order inductive type I (I is first-order if all constructor arguments are of first-order inductive type),
- $\Sigma\ ;\ []\ \vdash t : I\ a\_1 \dots a\_n$,
- t has eta-expanded constructor and fixpoint applications,
- $\Sigma\ ;\ []\ \vdash t \equiv v$
- v is irreducible
- p is the translation of t to a Malfunction program via erasure

then p evaluates to a value v' (containing closures) which unfolds to v

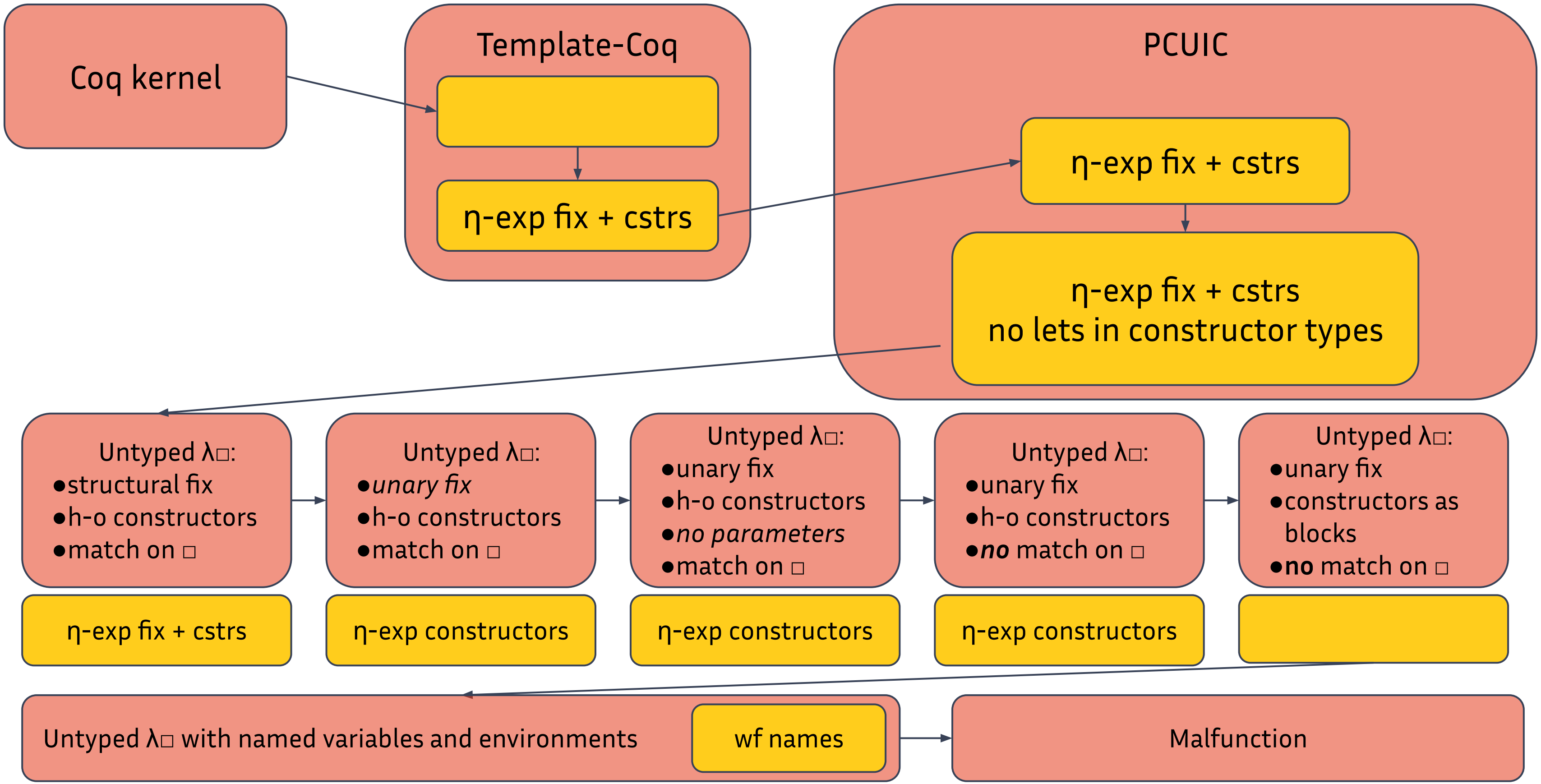
Verified Extraction to Malfunction

Coq / lambda box

- structural fix
- higher-order constructors
- match on □
- de Bruijn
- fix / match

OCaml / Malfunction

- unary let rec
- constructors are blocks
- cannot match on functions
- named
- let rec / switch / proj


Coq kernel
Template-Coq
η-exp fix + cstrs
PCUIC
η-exp fix + cstrs
η-exp fix + cstrs
no lets in constructor types
Untyped λ□:
●structural fix
●h-o constructors
●match on □
Untyped λ□:
●*unary fix*
●h-o constructors
●match on □
Untyped λ□:
●unary fix
●h-o constructors
●*no parameters*
●match on □
Untyped λ□:
●unary fix
●h-o constructors
●***no*** match on □
Untyped λ□:
●unary fix
●constructors as blocks
●**no** match on □
η-exp fix + cstrs
η-exp constructors
η-exp constructors
η-exp constructors
Untyped λ□ with named variables and environments
wf names
Malfunction

Almost™ verified extraction to OCaml

Corollary: Given

- k + 1 many first-order inductive types I_k
- Σ ; [] ⊢ t : I_1 a_1_1 ... a_1_n1 -> ... -> I_k a_k_1 ... a_k_nk
- t has eta-expanded constructor and fixpoint applications,
- p is the translation of t to a Malfunction program via erasure

then for all Malfunction terms x_1 ... x_k which terminate with normal form corresponding to a constructor application C_i args_i fitting into the type I_i, p x_1 ... x_k evaluates to a value unfolding to the value of t (C_1 args_1) ... (C_n args_n).

Why first-order inductives?

- Standardisation only holds like that, otherwise we'd need to talk about observational equivalence (type-based, even though the calculus is untyped)
- the erasure theorem for the erasure function only holds like that, otherwise there may be non-erased residues and we have to talk relationally everywhere

Free results

- The CertiCoq project (verified extraction from Coq to C) can benefit from our transformations
- Given a semantics of CakeML in Coq, we get operationally correct extraction to CakeML as well

Todo-Lists

Coq:

- replace template polymorphism with e.g. sort polymorphism

MetaCoq:

- Eta: Specify eta conversion and adapt checker
- Modules: Quoting to Template-Coq, typing, flattening to PCUIC

Malfunction:

- Add support for `Extract Inductive` and `Extract Constant`
- Add typing (realizability for Malfunction and OCaml types)
- Add axioms (realizability for $\lambda\square$ and Coq types)
- Add GADTs (realizability for Malfunction and OCaml types + GADTs)

Optimisations

- the optimisations currently used are not proved correct, not even on paper
- for some, it is clear what the theorem should be
- for others, the theorem will rely on observational equivalence
- working with Kazuhiko Sakaguchi (research engineer in Gallinette)
- primitive integers and floats?

Interfacing with OCaml code

“extracted programs don’t go wrong”?

```
safeHead : forall xs : list nat, xs <> nil -> nat
safeHead : list nat -> nat
safeHead []
```

in Pierre Letouzey’s thesis the theory is built up using logical relations again in the object theory

Letouzey's "semantic" correctness proof

Theorem: If $\Sigma; [] \vdash t : A$ then there is a proof of $\Sigma; [] \vdash p : [A]\ [t]\ \ t$
where [t] is translation of t to inductive representation of terms, and [A] is a relation.

Example:

$[\mathbb{N}]\ s\ n := \Sigma; [] \vdash [s \rightsquigarrow^* n]$
where $[s \rightsquigarrow^* n]$ is an inductive type in the object theory

$[\mathbb{N} \to \mathbb{N}]\ s\ f$: There is a proof term p proving

$$\Sigma; [] \vdash \forall\, s'\, n,\ [\mathbb{N}]\ s'\ n \to [\mathbb{N}]\ (s\ s')\ (f\ n)$$

i.e.

$$\Sigma; [] \vdash \forall\, s'\, n,\ [s \rightsquigarrow^* n] \to [s\ s' \rightsquigarrow^* f\ n]$$

Reduction quotation lemma

Theorem: If $\Sigma; [] \vdash s \leadsto^* t$ then there is a proof term p with $\Sigma; [] \vdash p : [s \leadsto^* t]$ where $[s \leadsto^* t]$ is an inductive encoding of reduction in the object theory.

Meta-level equivalent: If s reduces to t in Coq, then we can actually prove the MetaCoq statement $\Sigma; [] \vdash s \leadsto^* t$

Proof: By induction on the normal form of $\Sigma; [] \vdash s \leadsto^* t$.

Excursus: Church's thesis

CT := forall f : nat -> nat, exists t : Turingmachine, t computes f

This axiom is consistent in CIC via a (set-theoretic) model based on assemblies

It is an open question whether CT is consistent in MLTT

One approach: Provide not a set-theoretic, but an SN based proof of consistency via realizability-like semantics

Types of truth in proof engineering

1. “This theorem is true.”
2. “There is a proof of this theorem.”
3. “The proof of this theorem can be formalised.”
4. “The proof of this theorem can be formalised in less than a week.”

2, 3, and 4 need lots of experience to distinguish

Our estimates how long things will take are usually wrong

Alternative approach

Keep the relation [A] on the meta-level, i.e. prove

If $\Sigma ; [] \vdash t : A$ then $[A]\ \Sigma\ [t]\ \ t$
where [t] is translation of t to inductive representation of terms, and [A] is a relation:

$[\mathbb{N}]\ s\ n := \Sigma ; [] \vdash s \rightsquigarrow^* n$

$[\mathbb{N} \to \mathbb{N}]\ s\ f$: There is a proof term p proving

$\forall\ s'\ n,\ \ \Sigma ; [] \vdash [\mathbb{N}]\ s'\ n \to \Sigma ; [] \vdash [\mathbb{N}]\ (s\ s')\ (f\ n)$

i.e.

$\forall\ s'\ n, \Sigma ; [] \vdash s \rightsquigarrow^* n \to \Sigma ; [] \vdash s\ s' \rightsquigarrow^* f\ n$

How to define [I] for an inductive type I in general? For recursive occurrences, this will not work…

Step-indexing? Iris to the rescue?

What about effectful programs?
What about erased pre-conditions?

Infrastructure is important

200 k LoC, nested mutual inductive propositions with 25+ constructors

MetaCoq exposes the deficits of Coq for “real world” proof engineering:

- compilation is slow
- GUIs are suboptimal
- automatic generation of induction lemmas etc often fails us
- long-term maintainability of proofs is an issue
- reproducibility and forward CI are issues (much better already)

Consistency and strong normalization

- strict positivity checker for inductive is implemented
- guard condition for fixpoints are specified as syntactic oracles which must be preserved by reduction and substitution and ensure strong normalisation
- Touching proof-theoretic principles quickly: CIC seems to be the weakest constructive higher-order system on usual scales

Future work with Lennard Gäher

- specify the guard condition logically
- implement the guard condition
- reduce guard condition as is to more strict conditions stepwise

Modularity and meta-programming

- MetaCoq would benefit from modularity
- We don't have good approaches for modularity
- One approach: Coq à la Carte (jww Kathrin Stark), relying on lots of automation and meta-programming
- But: We don't have good interfaces for meta-programming
- MetaCoq could fill this gap, but lots of work is needed
- Tactics for modular programming?
- automatic demodularisation?
- Parametric transport

Towards Verified Extraction from Coq to OCaml

Give a verified implementation of extraction

- formalise Coq in Coq
- formalise Malfunction
- re-implement extraction
- verify it, operationally

TCB:

- the Malfunction & OCaml compilers
- Coq

TSB: Formalisation of CIC & Malfunction

`https://metacoq.github.io`