A certifying extraction with time bounds from Coq to call-by-value \( \lambda \)-calculus

Yannick Forster  Fabian Kunze

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Germany

ITP 2019
September 12
Formalising computability theory:

- Functions definable in constructive type theory are computable, but this is not provable \textit{inside} the theory
- Explicit model of computation needed for negative results or complexity theory
- We use: call-by-value $\lambda$-calculus “L”
- Computability provable for every \textit{concrete} function defined in Coq
- But computability proofs tedious and repetitive!

We provide a framework that automates\textsuperscript{1} this \textit{extraction}

\textsuperscript{1}for ML-like subset of Coq
Idea

Call-by-Value lambda calculus (Plotkin [1975], Forster and Smolka [2017]):

\[
s, t ::= \ x \mid \lambda \ s \mid st
\]

- For each concrete function \( f : X_1 \to \ldots \to X_n \) over ML-like data types,
- find a \( \lambda \)-term \( t_f \) and a time complexity \( \tau_f \) such that
- for all \( \vec{x} \), \( t_f(\text{enc}\vec{x}) \) reduces to \( \text{enc}(f\vec{x}) \)
- within \( \tau_f \vec{x} \) many beta-reduction steps (Accattoli and Lago [2016]).

Simple, syntax directed extraction process.
Example

```
Require Import LBool LTactics ComputableTactics.

Definition orb := \x \ y : \ -> if \x then true else \y.

Instance comp_orb : computable orb.
Proof.  
  extract.
Defined.

Eval cbv in (ext_orb).
```

= lam (lam (((lam (lam 1)) 0)) : extracted orb
Example with time

Require Import LBool LTactics ComputableTactics.

Definition orb := \! x y : \# \to if x then true else y.

Instance comp_orb : computableTime orb
  (\! \_ \_ \to (1, \! \_ \_ \to (3,tt))).

Proof.
  extract. solverec.
Qed.
Example with recursive function

Require Import LNat Nat LBool LTactics ComputableTactics.

Fixpoint add (n m : N) {struct n} : N :=
  match n with
  | 0 ⇒ m
  | S p ⇒ S (add p m)
  end.

Instance comp_add :
  computableTime add
  (λ n _ ⇒ (S, (λ m _ ⇒ (11*n+4, tt))))).

Proof.
  extract. solverec.
Qed.
Example with higher order function

```
Require Import Ltac Datatypes.Lists ComputableTactics.

Definition map (A B : Type) (f : A -> B) :=
  fix map (l : list A) : list B :=
    match l with
      | [] => []
      | a :: t => f a :: map t
    end.

Lemma term_map (X Y : Type) (Hx : registered X) (Hy : registered Y):
  computableTime (@map X Y)
    ((λ f fT => (1,
      λ l _ => (fold_right
        (λ x res => fst (fT x tt) + res + 11)
        7 l
      ,tt))).
  Proof.
    extract.
    solverec.
    Qed.
```
Outline

- What is correctness?
- How to extract terms?
- How to prove those extractions correct?
- Case studies:
  - Self-interpreter for L
  - Enumerator for diophantine equations (\(\rightarrow\) Reduction to L)
  - Turing machine interpreter
What is Correctness?
Encoding values: Scott encoding

Inductive values encoded as functions (corresponding to their `match`):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>$\lambda tf. t$</td>
</tr>
<tr>
<td>false</td>
<td>$\lambda tf. f$</td>
</tr>
<tr>
<td>0</td>
<td>$\lambda sz.z$</td>
</tr>
<tr>
<td>1</td>
<td>$\lambda sz. sz (\lambda sz.z)$</td>
</tr>
<tr>
<td>S</td>
<td>$\lambda sz. sz$</td>
</tr>
</tbody>
</table>

→ Encoding follows from definition of inductive data type
Generating encoding functions

Mechanically derivable\(^2\) using Template Coq, part of MetaCoq (Sozeau et al. [2019]):

\[
\text{enc_nat} = \begin{cases} 
\text{match x with} \\
0 & \Rightarrow \text{lam (lam 1)} \\
S\ x_0 & \Rightarrow \text{lam} \\
\end{cases}
\]

: encodable \(N\)

Stored in Typeclass, with properties (e.g. injectivity) proven by \texttt{Ltac}.

\(^2\)for all ML-like (non-dependent, non-mutual) inductive types
Correctness predicate

When does a (\(\lambda\)-)term \(t_f\) compute a (Coq-)function \(f\)?

Example: \(t_{orb} \sim orb:\)

\[ \forall xy : \mathbb{B}, t_{orb} (\text{enc} x) (\text{enc} y) \preceq^* \text{enc}(\text{orb} x y) \]

Tool: Logical relation!
Correctness predicate

When does a ($\lambda$-)term $t_f$ compute a (Coq-)function $f$?

Example: $t_{\text{orb}} \sim \text{orb}$:

$$\forall xy : B, t_{\text{orb}} (\text{enc} x) (\text{enc} y) \succ^* \text{enc}(\text{orb} x y)$$

Tool: Logical relation!

$$\frac{\text{enc}_A a \sim a \quad \text{(for } a : A)}{\text{enc}_A a \sim a}$$

$$\forall (a : A)(t_a : T). \quad t_a \sim a \quad \frac{t_f t_a \sim fa \quad \text{(for } f : A \to B)}{t_f \sim f}$$
Correctness predicate

When does a ($\lambda$-)term $t_f$ compute a (Coq-)function $f$?

Example: $t_{\text{orb}} \sim \text{orb}$:

$$\forall xy : B, t_{\text{orb}} (\text{enc}x) (\text{enc}y) \succ^* \text{enc}(\text{orb} \times y)$$

Tool: Logical relation!

\[
\begin{align*}
\text{enc}_A a & \sim a \quad \text{(for } a : A) \\
\hline
\text{enc}_A a & \sim a \\
\hline
\end{align*}
\]

$t_f$ is closed value $\land$

\[
\forall (a : A)(t_a : T). \ t_a \sim a \rightarrow \sum (v : T). \ t_f t_a \succ^* v \land v \sim fa \\
\hline
\end{align*}
\]

$tf \sim f$ \quad \text{(for } f : A \rightarrow B)
Correctness predicate

When does a (\(\lambda\)-)term \(t_f\) compute a (Coq-)function \(f\)?

Example: \(t_{\text{orb}} \sim \text{orb}\):

\[
\forall xy : \mathbb{B}, t_{\text{orb}} (\text{enc} x) (\text{enc} y) \leadsto^{*} \text{enc}(\text{orb} \times y)
\]

Tool: Logical relation!

\[
\begin{align*}
\text{enc}_A a & \sim a \quad \text{(for } a : A) \\
t_f \text{ is closed value } & \land \\
\forall (a : A)(t_a : \mathcal{T}). \ t_a & \sim a \rightarrow \Sigma (v : \mathcal{T}). \ t_f t_a \leadsto^{*} v \land v & \sim f a \quad \text{(for } f : A \rightarrow B) \\
\end{align*}
\]

Problem: Not strictly positive.
Solution: Recursion on Type...?
Correctness predicate (2)

Inductive $\mathcal{T} : Type \rightarrow Type :=$
    $\mathcal{T}_\text{base} A \{\text{registered A}\} : \mathcal{T} A$
  $| \mathcal{T}_\text{arr} A B (\tau_1 : \mathcal{T} A) (\tau_2 : \mathcal{T} B) : \mathcal{T} (A \rightarrow B)$.

Fixpoint computes $\{A\} (\tau : \mathcal{T} A) : A \rightarrow T \rightarrow Type :=$
    match $\tau$ with
    $\mathcal{T}_\text{base} \Rightarrow \text{fun} x x\text{Int} \Rightarrow (x\text{Int} = \text{enc} x)$
    $| @\mathcal{T}_\text{arr} A B \tau_1 \tau_2 \Rightarrow$
    fun f t_f \Rightarrow
      proc t_f \Rightarrow \forall (a : A) t_a,
      computes $\tau_1 a t_a$
      \rightarrow $\{v : \text{term} \& (t_f t_a >* v) \ast \text{computes} \tau_2 (f a) v\}$
    end%type.
**Complexity functions**

Describe number of steps dependent on input.

Example:

\[
\begin{align*}
\text{orb } & \text{true false} \\
(\text{fun } & x \ y : \text{bool } \Rightarrow \text{if } x \text{ then true else y}) \text{ true false} \\
(\text{fun } & y : \text{bool } \Rightarrow \text{if } \text{true then true else y}) \text{ false} \\
(\text{if } & \text{true then true else false}) \text{ true}
\end{align*}
\]

\[\Rightarrow \text{Time described by function}\]

\[
\tau_{\text{orb}} : \mathbb{B} \to \mathbb{N} \times (\mathbb{B} \to \mathbb{N}) := \lambda x.(1, \lambda y.2)
\]
Complexity functions (2)

- Recursive functions:

\[ \tau_+ : \mathbb{N} \rightarrow \mathbb{N} \times (\mathbb{N} \rightarrow \mathbb{N}) := \lambda m.(c_1, \lambda n.m \cdot c_2 + c_3) \]

- Higher-order functions: Type for time-complexity functions actually more involved, see paper for full truth.

- Correctness predicate easily extended with time-complexity functions:

\[ t_f \sim_{\tau_f} f \]
Extracting Functions
Extracting functions

- Variables translate to variables according to environment
- Abstraction translates to lambda
- Fixpoints translate to fixed-point combinator $\rho$

```
Fixpoint extract (env : nat → nat) (s : Ast.term) (fuel : nat) :
  TemplateMonad T :=
mismatch fuel with
  0 ⇒ tmFail "out of fuel" | S fuel ⇒
mismatch s with
  Ast.tRel n ⇒ ret (var (env n))
| Ast.tLambda _ _ s ⇒
    t ← extract (↑ env) s fuel ;;
    ret (lam t)
| Ast.tFix [Ast.mkdef _ _ s_]_ ⇒
    t ← extract (↑ env) s fuel ;;
    ret ($\rho$ (lam t))
```
Extracting functions (2)

- Applications are application
- Extractions of constants get reused using Typeclasses

| Ast.tApp s R ⇒ params ← tmDependentArgs s;; if params =? 0 then
| t ← extract env s fuel;; monad_fold_left (fun t1 s2 ⇒ t2 ← extract env s2 fuel ;; ret (app t1 t2)) R t
| else let (P, L) := (firstn params R,skipn params R) in
| s’ ← tmEval cbv (Ast.tApp s P);; a ← tmUnquote s’;; a’ ← tmEval cbn (my_projT2 a);; nm ← (tmEval cbv (String.append (name_of s) "_term") >>= tmFreshName);; i ← tmTryInfer nm (Some cbn) (extracted a’) ;; let t := (@int_ext _ _ i) in
| monad_fold_left (fun t1 s2 ⇒ t2 ← extract env s2 fuel ;; ret (app t1 t2)) L t
Proving correctness
Proving correctness

Each extracted term is certified using $\text{Ltac}$:
- Tactic reducing $\lambda$-terms keeping track of number of steps
- Show correctness by following same case distinctions/recursions as used in the extracted function
Proving correctness, example

```
Require Import Ltactics Datatypes.Lists ComputableTactics.
Import Intern.

Definition map (A B : Type) (f : A → B) :=
  fix map (l : list A) : list B :=
  match l with
  | [] ⇒ []
  | a ++ t ⇒ f a ++ map t
  end.

Lemma term_map (X Y:Type) (Hx : registered X) (Hy:registered Y):
  computable (@map X Y).
Proof.
  extract.
Qed.
```
Proving correctness, example

```
Require Import L Tactics Datatypes.Lists ComputableTactics.
Import Intern.

Definition map (A B : Type) (f : A → B) :=
  fix map (l : list A) : list B :=
  match l with
  | [] ⇝ []
  | a :: t ⇝ f a :: map t
  end.

Lemma term_map (X Y:Type) (Hx : registered X) (Hy:registered Y):
  computable (map (B:=Y)).

Proof.
  extractAs s. computable_using_noProof s.
  cstep.
  cstep.
  allcstep.
Qed.
```
Proving correctness, example

\begin{verbatim}
Require Import Ltactics Datatypes.Lists ComputableTactics.
Import Intern.

Definition map (A B : Type) (f : A → B) :=
  fix map (l : list A) : list B :=
  match l with
  | [] ⇒ []
  | a :: t ⇒ f a :: map t
end.

Lemma term_map (X Y:Type) (Hx : registered X) (Hy:registered Y):
  computable (@map X Y).
Proof.
  extractAs s. computable_using_noProof s.
cstep.
cstep.
cstep.
allicstep.
Qed.
\end{verbatim}
Proving correctness, example

```
Require Import LTactics Datatypes.Lists ComputableTactics.

Import Intern.

Definition map (A B : Type) (f : A -> B) :=
  fix map (l : list A) : list B :=
  match l with
  | [] => []
  | a :: t => f a :: map t
  end.

Lemma term_map (X Y:Type) (Hx : registered X) (Hy:registered Y):
  computable (@map X Y).
Proof.
  extractAs s. computable_using_noProof s.
  cstep.
  cstep.
  all:cstep.
Qed.
```

```
X : Type
Y : Type
Hx : registered X
Hy : registered Y
s := (lam (rho (lam (lam ((O (int_ext []])) (lam (lam ((
    extracted (map (B:=Y)) : extracted (map (B:=Y))
computes ((! X -> ! Y) -> ! list X -> ! list Y)
  (lam (rho (lam (lam ((O (ext []))) (lam (lam ((
```

Yannick Forster, Fabian Kunze
Certifying extraction with time bounds
ITP 2019, September 12
Proving correctness, example
Proving correctness, example
Proving correctness, example

```
Require Import LTactics Datatypes.Lists ComputableTactics.
Import Intern.

Definition map (A B : Type) (f : A → B) :=
  fix map (l : list A) : list B :=
  match l with
  | [] → []
  | a :: t ⇒ f a :: map t
end.

Lemma term_map (X Y:Type) (Hx : registered X) (Hy:registered Y):
  computable (@map X Y).
Proof.
  extractAs s. computable_using_noProof s.
cstep.
cstep.
all:cstep.
Qed.
```
Proving correctness, example

```
Require Import LtacLtactics Datatypes.Lists ComputableLtactics.
Import Intern.

Definition map (A B : Type) (f : A → B) :=
fix map (l : list A) : list B :=
match l with
| [] → []
| a :: t → f a :: map t
end.

Lemma term_map (X Y:Type) (Hx : registered X) (Hy:registered Y):
  computable (@map X Y).
Proof.
  extract.
Qed.
```
Deriving time complexity
Solving recurrence relations interactively

Require Import LTactics Datatypes.Lists ComputableTactics.

Definition map (A B : Type) (f : A → B) :=
  fix map (l : list A) : list B :=
  match l with
  | [] → []
  | a :: t → f a :: map t
  end.

Lemma term_map (X Y:Type) (Hx : registered X) (Hy:registered Y):
  computableTime (@map X Y)
  (λ f fT → (1,
    λ l_ → (fold_right
    (λ x res ⇒ fst (fT x tt) + res +
    7 l,
    tt))).

Proof.
extract.
solverec.
Qed.

Yannick Forster, Fabian Kunze
Certifying extraction with time bounds
ITP 2019, September 12
Deriving time complexity
Solving recurrence relations interactively

Require Import LTactics Datatypes.Lists ComputableTactics.

Definition map (A B : Type) (f : A → B) :=
fix map (l : list A) : list B :=
match l with
| [] => []
| a :: l => f a :: map l
end.

Lemma term_map (X Y:Type) (Hx : registered X) (Hy:registered Y):
  computableTime (@map X Y)
  (λ f fT ⇒ (cnst "c",
       λ l _ ⇒ (cnst ("f", l),tt))).

Proof.
  extract.
  solverec. 3:rename xT into x_f_, x_2 into a_, l into l_;
  Qed.

X : Type
Y : Type
Hx : registered X
Hy : registered Y
used_term := lam (rho (lam (lam ((O (int_ext [])) (lam (lam (((int_ext cons) (S 1)) (S 0))))))))
    : extracted (map (B:=Y))

x : X → Y

xt : X → unit → N × unit

1 ≤ cnst "c"

subgoal 2 (ID 7179) is:
7 ≤ cnst ("f", [ ])

subgoal 3 (ID 8208) is:
fst (x_f_ a tt) + cnst ("f", l) + l ≤ cnst ("f", a :: l)

Yannick Forster, Fabian Kunze
Certifying extraction with time bounds
ITP 2019, September 12

22
Deriving time complexity
Solving recurrence relations interactively

Yannick Forster, Fabian Kunze
Deriving time complexity
Solving recurrence relations interactively
Case Study
Library of datatypes and functions

Build upon shared extraction of:

- \( \mathbb{B} \), options, pairs . . .
- \( \mathbb{N} \): addition, multiplication, equality . . .
- Lists: map, filter, . . .
- \( \approx \) 360 lines
Self interpreter for L

Universal L-term:

- Function $\text{eva} : \mathbb{N} \to T \to T_\perp$ with $s \rhd^* t \iff \exists n, \text{eva} n s = t$
- Base for many results in computability theory
- 20 lines from specification in Coq to correct extraction in L
Self interpreter for L

```
(** Evaluation as a function *)

Fixpoint eva (n : N) (u : term) :=
  match u with
  | var n ⇒ None
  | lam s ⇒ Some (lam s)
  | app s t ⇒ match n with
    | 0 ⇒ None
    | S n ⇒ match eva n s, eva n t with
      | Some (lam s), Some t ⇒ eva n (subst s 0 t)
      | _, _ ⇒ None
  end
end.
```

Yannick Forster, Fabian Kunze  Certifying extraction with time bounds  ITP 2019, September 12
Self interpreter for L

```coq
(** ** Encoding for L-terms **) Run TemplateProgram (imGenEncode "term_enc" term). Hint Resolve term_enc_correct : Lrewrite. (** register the non-constant constructors **) Instance term_var : computableTime var (\lambda n . \equiv (l, tt)). Proof. extract constructor. solverec. Qed. Instance term_app : computableTime app (\lambda s1 \Rightarrow (l, \lambda s2 \Rightarrow (l, tt)\equiv ))). Proof. extract constructor. solverec. Qed. Instance term_lam : computableTime lam (\lambda s . \equiv (l, tt)). Proof. extract constructor. solverec. Qed; (** ** Extracted equality on natural numbers **) Instance term_nateqb : computableTime Nat.eqb (\lambda x xT \equiv (5,(\lambda y yT \equiv (\lambda (\min x y)\times 15 + 8,tt))). Proof. extract. solverec. Qed. (** ** Extracted substitution **) Instance term_substT : computableTime subst (\lambda s . \equiv (5, (\lambda n . \equiv (l, (\lambda t . \equiv (15 \times n \times size s + 43 \times (size s) \times 2 + 13, tt)))))). Proof. extract. solverec. Qed. (** ** Extracted step-indexed L-interpreter **) Instance term_eva : computable eva. Proof. extract. Qed.
```
Self interpreter for L

Yannick Forster, Fabian Kunze
Certifying extraction with time bounds
ITP 2019, September 12
Self interpreter for L

```plaintext
(* * Encoding for L-terms *)
Run TemplateProgram (imGenEncode "term_enc" term).
Hint Resolve term_enc_correct : Lrewrite.

(* register the non-constant constructors *)
Instance term_var : computableTime var (λ n → (1, tt)).
Proof. extract constructor. solvereq. Qed.

Instance term_app : computableTime app (λ s₁ → (1, (λ s₂ → (1, tt))))
(λ s → (1, tt))).
Proof. extract constructor. solvereq. Qed.

Instance term_lam : computableTime lam (λ s → (1, tt)).
Proof. extract constructor. solvereq. Qed.

(* * Extracted equality on natural numbers *)
Instance term_nat_eqb : computableTime Nat.eqb (λ x y → (5, (λ y y → (θ
(min x y) × 15 + 8, tt))))).
Proof. extract. solvereq. Qed.

(* * Extracted substitution *)
Instance term_substT : computableTime subst (λ s → (5, (λ n → (1, (λ t →
(size s + 43 × (size s) ^ 2 + 13, tt)))))).
Proof. extract. solvereq. Qed.

(* * Extracted step-indexed L-interpreter *)
Instance term_eva : computable eva.
Proof. extract. Qed.
```
Diophantine equations

- Diophantine sets are recursively enumerable
- \[ \Rightarrow \] Many-one reduction from diophantine equations to L-halting problem
- \( \approx 200 \) lines
Turing machine interpreter

- Each step of a TM can be computed in constant time in L:

  \[
  \text{Global Instance } \text{term\_loopM} : \\
  \text{let } c1 := (\text{haltTime} + n*121 + \text{transTime} + 76) \text{ in } \\
  \text{let } c2 := 13 + \text{haltTime} \text{ in } \\
  \text{computableTime (loopM (M:=M))} \\
  \hspace{1cm} (\text{fun } \_ \_ \Rightarrow (5, \text{fun } k \_ \Rightarrow (c1 * k + c2, tt))).
  \]

- Many-one reduction from TM-halting to L-halting problem
- \(\approx 400 \) lines
Future Work

- Formalise complexity theory:
  - Extract *efficient* self interpreter
  - Time Hierarchy Theorem
  - NP-Completeness
- Include space analysis in framework
- Verify extraction process using MetaCoq
- Better treatment of dependent functions (realisability model for Coq)
Related Work

- Myreen and Owens: HOL4 to CakeML
- Hupel and Nipkow: Isabelle/HOL to CakeML
- Köpp: Minlog to $\lambda$-Calculus
- Œuf project: Verified compiler from Coq-Subset to Assembly
- CertiCoq project: Verified extraction from Coq to Clight.
Contribution

- A plugin extracting Coq functions of simple polymorphic types to cbv \( \lambda \)-calculus L
- Logical relations connecting Coq functions with correct extractions and time bounds
- Automated correctness and semi-automated time analysis for extracted terms
- Three case studies and library including \( \mathbb{N} \) and lists:
  - Universal L-term
  - Reduction from Diophantine equations to L-halting problem
  - Polynomial-time simulation of Turing machines in L
- Contributed to library of undecidable problems in Coq: github.com/uds-psl/coq-library-undecidability

Thank you!
References


<table>
<thead>
<tr>
<th>Framework:</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness predicate</td>
<td>370</td>
</tr>
<tr>
<td>Extraction</td>
<td>380</td>
</tr>
<tr>
<td>Simplification Tactics</td>
<td>950</td>
</tr>
<tr>
<td>Verifying Tactics</td>
<td>420</td>
</tr>
<tr>
<td>total</td>
<td>2100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case Studies:</th>
<th>spec</th>
<th>proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library</td>
<td>355</td>
<td>282</td>
</tr>
<tr>
<td>Universal Term</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>H10</td>
<td>79</td>
<td>124</td>
</tr>
<tr>
<td>Universal TM</td>
<td>243</td>
<td>151</td>
</tr>
</tbody>
</table>
Correctness with time bounds

\[
\begin{align*}
\text{enc}_A a & \sim^\tau a \quad \text{(for } a : A) \\
t_f \text{ is a procedure } & \wedge \\
\forall at_{aT_a}. \ t_a \sim^{\tau_a} a \rightarrow \Sigma \nu : T. \\
\phantom{t_f} & \geq^n \nu \land \nu \sim^\tau fa \text{ where } \tau_f a_{\tau a} = (n, \tau) \\
\end{align*}
\]

\[
\begin{align*}
t_f & \sim^{\tau_a} f \quad \text{(for } f : A \rightarrow B) \\
\end{align*}
\]