Verified Extraction from Coq to a Lambda-Calculus

Yannick Forster

joint work with Fabian Kunze

28.08.2015

Abstract

We present a framework to export programs in Coq to a weak call-by-value lambda calculus called L. L can be seen as a very basic functional programming language, featuring abstraction, match-constructs based on Scott's encoding and full recursion.
The extraction from Coq is not verified itself, but produces proofs for the correctness of each single extracted program semi-automatically. The frameworks builds on the Coq plugin Template Coq by Gregory Malecha [4]. It eliminates the non-computational parts of the Coq program and produces a term in L and the corresponding correctness statement, which in turn can be verified using several provided automation tactics (developed in [3]).
Using the framework, one can focus on the interesting parts of developments in L, because the task of programming is taken care of. We use the framework to develop a formalization of Computability theory [2] as a case study.

1 Overview

This report is intended to be an overview of the basic ideas behind a framework rather than a documentation or a manual. It tries to be mostly self contained. The framework itself grew out of [2] and heavily depends on the automation developed in [3].

We use Template Coq to get an inductive representation from Coq terms. We directly translate this representation to our own intermediate representation, which is more concise and gives the possibility to eliminate non computational parts. We describe this in in section 2.

In sections 3.1 and 3.2 we give a short introduction to L and Scott's encoding and describe the mathematical idea behind the framework. The implementation in Coq using type classes is described in sections 3.3 and 3.4.

Section 4.1 deals with the generation of correctness statements and 4.2 gives a short overview of the semi-automatic proofs.

Finally, we reimplement the development of basic computability theory from [2] and compare the two formalizations in section 5.

2 Representations for Coq terms

2.1 Template Coq

Template Coq [4] "is a quoting library for Coq. It takes Coq terms and constructs a representation of their syntax tree as a Coq inductive data type"[1] developed by Gregory Malecha.

[1]cited from `https://github.com/gmalecha/template-coq/`

We use tactics provided by Template Coq to convert a Coq term to an inductive representation. We give a short overview over this inductive representation and briefly describe our contributions to Template Coq.

```
Inductive term : Type :=
| tRel : ℕ→ term
| tVar : ident → term (** this can go away **)
| tMeta : ℕ→ term (** NOTE: this can go away *)
| tEvar : ℕ→ term
| tSort : sort → term
| tCast : term → cast_kind → term → term
| tProd : name → term (** the type **) → term → term
| tLambda : name → term (** the type **) → term → term
| tLetIn : name → term (** the type **) → term → term → term
| tApp : term → list term → term
| tConst : string → term
| tInd : inductive → term
| tConstruct : inductive → ℕ→ term
| tCase : ℕ(* # of parameters *) → term (** type info **) → term → list term → term
| tFix : mfixpoint term → ℕ→ term
| tUnknown : string → term.
```

Because we are mainly interested in the computational part of Coq terms and especially in the extraction of closed terms, many of those constructors are uninteresting for us.

The most important part of Template Coq for our purposes is the tactic `quote_term k x`, that takes a tactic `k` (expecting one argument) and a term `x`. It then quotes the term `x` giving the result to the tactic `k`.

The inductive representation of `plus` (without any unfolding) is `Ast.tConst "Coq.Init.Nat.add"`. In many cases, we need to be able to reproduce the term `plus` out of the string `"Coq.Init.Nat.add"`. This was previously not provided by Template Coq as a tactic, but only as a Vernacular command. We added the tactic `denote_term k x` working inverse to `quote_term` for a restricted subset of the `Ast.term` type to Template Coq[2].

2.2 Intermediate Representation

The AST representation of Template Coq carries a lot of information which is unnecessary for the purpose of extraction. To ease for instance the elimination of propositions before the extraction, we define an intermediate representations, that specifies the subset of Coq which can be extracted. This can also be seen as a clean interface to extract to other languages than L.

```
Inductive iTerm : Prop :=
  iApp : iTerm → iTerm → iTerm (* application of two terms *)
| iLam : iTerm → iTerm (* fun *)
| iFix : iTerm → iTerm (* fix *)
| iConst (X:Type) : X → iTerm (* not unfolded constants *)
| iMatch : iTerm → list iTerm → iTerm (* matches with all the cases *)
| iVar : ℕ→ ℕ→ iTerm (* variables *)
| iType (X : Type) : X → iTerm. (* elimiℕed terms *)
```

Almost every constructor should be self-explanatory. The match constructor carries the list of its match branches, which are represented as abstractions. Variables have a second argument, that will be used to conveniently extract fixed points (see section 3.2.3). The `iType` and `iConst` constructors will be explained in the following sections.

[2]see `https://github.com/gmalecha/template-coq/commit/c6de18e0ba00980e82f61c98e2348c292f86faa5`

2.3 Eliminating Propositions

The `Extraction` command in Coq conveniently removes all proofs and computation on types from the terms and extracts the computationally relevant part only. This is especially useful in the case of deciders, for instance of type

```
∀ x y : ℕ, { x = y } + {x ≠ y }
```

which can be extracted to a term of type `ℕ→ ℕ→ bool` in the obvious way. We mimic this behavior. Currently, we eliminate non computational part by a heuristic, that works in all interesting cases. The key idea is that we compute the `typeLevel` of a Coq term `t` in Ltac. If `t : Type`, it has `typeLevel` 0. If `t : X → Y`, it has typeLevel $n+1$ if Y has `typeLevel` n.

We eliminate parts of the term that have `typeLevel` 0.

3 Extraction

3.1 A Weak Call-By-Value Lambda Calculus

We use a weak call-by-value lambda calculus called L, already used in the development of computability theory in [2].L is syntactically a lambda calculus with de-Bruijn variables and uses the following reduction rules:

$$\frac{}{(\lambda s)\ v \succ s^0_v} \qquad \frac{s \succ s'}{s\ t \succ s' t} \qquad \frac{t \succ t'}{st \succ st'}$$

We write $\succ^*$ for the reflexive, transitive closure of $\succ$ and $\equiv$ for the equivalence closure. Note that $\succ^*$ is a subrelation of $\equiv$ and $s \equiv \lambda t \;\leftrightarrow\; s \succ^* \lambda t$. For further properties of L see [2], chapters 2 and 3.

3.2 Scott's Encoding

There are several well-known ways to represent constructor types in the lambda calculus. One is a representation via Church's encoding, which does not work in the setting of call-by-value reduction.

A second way would be to encode the terms as natural numbers via some sort of Gödelization and encode this number, as it is often done in Computability theory. While this is technically easy, programming with such encoded terms is almost impossible.

Thus we use Scott's encoding [1], which was used for terms by Mogensen [5], similar to the development in [2]. Scott's encoding encodes a term t as the match-construct that is implicitly hidden in the term.

As an example, natural numbers are encoded via a recursive procedure $x \mapsto \overline{x}$ as follows:

$$\overline{0} := \lambda zs.z$$
$$\overline{Sn} := \lambda zs.s\ \overline{n}$$

One can now define the constructor S as a lambda term:

$$\ulcorner S \urcorner := \lambda xzs.s\ x$$

We say that $\ulcorner x \urcorner$ is the *extracted* version of the Coq term x. We also say that $\ulcorner x \urcorner$ *internalizes* the Coq term x. We will use both terminologies interchangeably.

We will from now on always use the notation $\overline{x}$ to refer to the Scott encoding of a Coq term x. Note that we have $\ulcorner 2 \urcorner = \ulcorner S \urcorner \; (\ulcorner S \urcorner \; \ulcorner 0 \urcorner) = \ulcorner S \urcorner \; (\ulcorner S \urcorner \; \overline{0}) \succ^* \overline{2}$. In general, we get $\ulcorner x \urcorner \succ^* \overline{x}$.

Arbitrary constructor datatypes can be encoded in a similar way:

Assume that the type X has n constructors $c_1, \dots, c_n$. For any k-ary constructor c_i an element $c_i \; x_1 \dots x_k$ is represented as

$$\lambda \; c_1 \dots \; c_n. \; c_i \; x_1 \dots x_k$$

3.2.1 Application, Abstraction, Variables

The subset of Coq consisting of applications, abstractions and variables is essentially a typed version of the pure λ-calculus. It thus is straightforward how to internalize those terms.

If $\ulcorner s \urcorner$ internalizes a term s and $\ulcorner t \urcorner$ internalizes a term t then $\ulcorner s \urcorner \ulcorner t \urcorner$ internalizes the application $s \; t$. On a high level we see that always $\ulcorner s \urcorner \; \ulcorner t \urcorner \succ^* \ulcorner s \; t \urcorner$.

Abstractions and variables are also straightforward in the obvious way.

3.2.2 Match

Internalizing `match` constructs is easy, because a Scott encoding is essentially nothing more than a match construct.

Now every term $\lambda \; c_1 \dots \; c_n. \; c_i \; x_1 \dots x_k$ yields a match construct. The Coq-match

```
match t with
  | c_1 x_1 ... x_{k_1} ⇒ f_1  x_1 ... x_{k_1}
  | ...
  | c_n x_1 ... x_{k_n} ⇒ f_n  x_1 ... x_{k_n}
end
```

is simply done with $t \; f_1 \; \dots \; f_n$.

Note that this only works for strongly normalizing terms, which is no problem in our setting, because every term in Coq is strongly normalizing and extraction will preserve this property.

3.2.3 Fix

Fix is technically the hardest part. In contrast to [2] we do not use a fixed point combinator here. For a well behaving fixed point combinator R, we would like to have the property that $Ru \equiv u \; (R \; u)$ for a procedure u. This seems impossible in a call-by-value setting and the best equation one can get is $Ru \equiv \lambda x.u \; (R \; u) \; x$. Even more problematic, a term Ru is no procedure anymore (because it is an application). The last can be solved by the definition of a function $\rho : \mathbf{T} \to \mathbf{T}$ following [6].

We define:

$$A := \lambda zg.g(\lambda x.zzgx)$$
$$\rho s := \lambda x..A \; A \; s \; x$$

and have that $(\rho \; u) \; t \equiv u \; (\rho \; u) \; t$ for procedures u and t.

A technical difficulty comes up when dealing with nested fixed-points or functions of the form `fun x ⇒fix f y :=...`. The introduction of the additional λx in ρ requires us to essentially increase every de-Bruijn index following to a `fix` by one. We do this by annotating every variable with the corrected de-Bruijn index. This is because we will need the initial index as well as the corrected one later.

```
Ltac annotateVariables' t L :=
  let f x :=annotateVariables' x L in
  match eval hnf in t with
  | iVar ?n ?m ⇒
    let m' :=eval lazy in (m + elAt' L m) in
    constr: (iVar n m')
  | iApp ?s ?t ⇒
    let s' :=annotateVariables' s L in
    let t' :=annotateVariables' t L in
    constr: (iApp s' t')
  | iLam ?s ⇒
    let s' :=annotateVariables' s (0 :: L) in
    constr: (iLam s')
  | iFix ?s ⇒
    let s' :=annotateVariables' s (0::map S L) in
    constr: (iFix s')
  | iMatch ?s ?l ⇒
    let l' :=list_map iTerm f l in
    let s' :=annotateVariables' s L in
    constr:(iMatch s' l')
  | iType ⇒iType
  | iConst ?f ⇒constr:(iConst f)
  | _ ⇒fail 1000 "annotation failed in:" t
  end.

Ltac annotateVariables t :=annotateVariables' t (@nil ℕ).
```

The tactic maintains a list `L` of natural numbers. The m'th number in `L` stores the number of `fix` introduced between the binder and the variable m in the current context.

3.3 A Typeclass for Scott Encodings

On paper, we use the notation $\overline{2}$ or $\overline{true}$ to denote the Scott encoded terms 2 and *true*. In Coq, without further work we need to write for instance `ℕ_enc 2` and `bool_enc true`. The typeclass mechanism of Coq allows us to use overloading as on paper.

```
Class registered (X : Type) :=mk_registered
  {
    enc_f : X → term ; (* the encoding function for X *)
    proc_enc : ∀ x, proc (enc_f x) (* encodings need to be a procedure *)
  }.
Arguments enc_f X {registered} _.
```

A type now can be registered by giving an encoding function and a proof that this function yields procedures only. Note that this allows for more than Scott encodings. Our automation will nevertheless only work for Scott encodings.

We define a tactic to register encodings easily:

```
Ltac register encf :=refine (@mk_registered _ encf _);
  abstract (now ((induction 0 || intros);simpl;value)).
```

The `value` tactic is taken from [3].

Registration is straightforward now:

```
Instance register_bool : registered bool.
Proof.
  register bool_enc.
Defined.

Instance register_ℕ : registered ℕ.
Proof.
```

```
  register ℕ_enc.
Defined.
```

Note that we close the `Instance` with the `Defined` command to make `enc_f` available for computations. The proof remains opaque because we use the `abstract`[3] tactic.

The following definition makes the overloading possible:

```
Definition enc (X : Type) (H:registered X) : X → term :=enc_f X.
Global Arguments enc {X} {H} _ : simpl never.
```

We use a well-known trick here. Note that the function `enc` essentially is just a projection on one of its arguments. But, the only non-implicit argument is the third one. Especially, the argument `H`, from which the result is taken, is made implicit. Now when one writes `enc 2`, Coq derives (if possible) automatically the argument `H : registered ℕ`, takes the projection function and applies it to `2`.

Thus, the following computation works:

```
Compute (enc 0, enc false, enc 2).

= ((λ (λ 1)), (λ (λ 0)), (λ (λ 0 (λ (λ 0 (λ (λ 1)))))))
: term * term * term
```

3.4 A Typeclass and Tactics for Internalization

The translation to L-terms which was described mathematically in the last sections can be done by a short Ltac tactic. But we want to be able to reuse previously internalized subterms. This is why we employ the typeclass mechanism of Coq again. We introduce a typeclass `internalized` now, that will be extended in the following sections.

```
Class internalized (X : Type) (x : X) :=
{ internalizer : term ;
  proc_t : proc internalizer
}.

Definition int (X : Type) (x : X) (H : internalized x) :=internalizer.
Global Arguments int {X} {ty} x {H} : simpl never.
```

The key part again is the definition of `int`. It uses the same trick as the function `enc` before. After a function, say, `plus` is registered, one can write `int plus` to get the corresponding L-term.

We will now focus on how to use this class and the mathematically described ideas to write a tactic translating intermediate terms to lambda terms. We already explained every interesting non-technical problem but one. This appears in the context of lists and options, in general where polymorphic datatypes appear.

First, we define how to encode options:

```
Section Fix_X.
  Variable X:Type.
  Variable intX : registered X.

  Definition option_enc (t:option X) :term :=
  match t with
  | Some t ⇒λ(λ (1 (enc t)))
  | None ⇒λ(λ 0)
  end.

  Global Instance register_option : registered (option X).
```

[3]See `https://coq.inria.fr/refman/Reference-Manual011.html#hevea_tactic189`

```
  Proof.
    register option_enc.
  Defined.

  Global Instance term_None : internalized (@None X).
  Proof.
    ...
  Defined.

  Global Instance term_Some : internalized (@Some X).
  Proof.
    ...
  Defined.
End Fix_X.
```

We omit the proof scripts for now. Note that in order to generate correctness statements we have to internalize `@Some X` instead of `Some` only. Using this definitions the problem comes up when internalizing, say, the term `@Some ℕ`. After Prop/Type-elimination the term is represented as `iApp (iConst Some) (iType ℕ)`. One can now easily see that the usual approach does not work here. Internalizing `iType ℕ` will yield a dummy term, but internalizing `Some` does not work, because we have only registered `@Some ℕ`. Thus, our internalization tactic has to be a bit more complex. We first give the code and explain the interesting parts afterwards:

```
Ltac reconstruct t :=
  match t with
  | iApp ?s ?t ⇒
    let s' :=reconstruct s in
    let t' :=reconstruct t in
    constr: (s' t')
  | iType (Some ?x) ⇒constr:(x)
  | iType None ⇒constr:(True)
  | iConst ?f ⇒constr:(f)
  | _ ⇒fail 1000 "reconstruction failed for:" t
  end.

Ltac toLambda t (* : iTerm → L.term + Coq.term *) :=
  match t with
  | iVar ?n ?m ⇒let v :=eval cbv in (@inl term True (Lvw.var m)) in constr: v
  | iLam ?t ⇒
    match toLambda t with
    | inl ?x ⇒constr : (@inl term True (λ x ))
    end
  | iApp ?s ?t ⇒
    let x :=toLambda s in
    match x with
    | inl ?r1 ⇒
      let y :=toLambda t in
      match y with
        inl ?r2 ⇒constr: (@inl term True (app r1 r2))
      | inr _ ⇒fail 1000 "internalized lhs" s "succesfully, but rhs " t "failed"
      end
    | inr ?r1 ⇒
      let r2 :=reconstruct t in
      constr:( @inl term True (int (r1 r2)) )
    (* if internalization failed *)
    | inr ?r1 ⇒
      let r2 :=reconstruct t in
      constr: (@inr True _ (r1 r2))
    | inr ?r1 ⇒
     fail 1000 "internalization failed for" r1 "(reconstructing" t "didn't work)"
    end
  | @iConst ?X ?x ⇒
    constr: (@inl term True (int x))
  | @iConst ?X ?x ⇒
    constr: (@inr True X x)
  | iFix ?s ⇒
    match toLambda s with
      inl ?s' ⇒constr:(@inl term True(Lvw.rho (Lvw.λ s')))
    | _ ⇒fail 100 "failed under fix" t
    end
  | iMatch ?s ?l ⇒
    let l' :=matchToApp s l in
    let t :=toLambda l' in
    constr:(t)
  | iType _ ⇒constr: ( @inl term True I)
  | _ ⇒fail 100 "failed to internalize iterm " t
  end.
```

The `toLambda` tactic takes an `iTerm` as argument. It then returns either an L-term, or, if that did not work, tries to reconstruct the Coq-term. The cases for variables, abstractions, fix and match are straightforward.

We explain the behavior on applications, constants and eliminated Types by means of an example.

On the argument `iApp (iConst @Some) (iType ℕ)`, it will first try to internalize `iConst @Some` in the usual way. Since this fails, it returns `inr @Some`, reconstructs the right hand side also (yielding simply ℕ), puts the two parts together and tries to internalize `@Some ℕ` afterwards, finally succeeding.

4 Automated Verification

Until now we described how to extract L-terms from Coq code. The clear structure of the extraction and Scott's encoding allow us now to formulate and prove correctness properties of the extracted terms highly automated.

Section 4.1 describes how we generate correctness statements. Section 4.2 the connection to the automation that can be used to generate the corresponding proofs.

4.1 Generating Correctness Statements

One can read the corresponding correctness lemma of a term from its type. The correctness lemma of `plus : ℕ→ ℕ→ ℕ` reads `∀ n m : ℕ, (int plus) (enc n) (enc m) >* enc (plus n m)`.

One choice would be now to generate the correctness lemma via a tactic and store it in the instance of the `internalized` class. This would be easy to work with, but bear the risk that trivial correctness lemmas get used. Thus proof reading Coq proofs generated by our framework would be considerably more complicated, because the statement that a term was correctly internalized is based on the concrete correctness statement.

We use a different but similar approach. First, we give an inductive datatype `TT X` which inductively represents the type `X`. We then define a function `internalizesF : TT X → Prop` that generates the correctness lemma for each proof by structural recursion over the `TT` object. While proof reading, one simply needs to check that `internalizesF` is well-behaving, every other statement then can be trusted.

```
Inductive TT : Type → Type :=
  TyB t (H : registered t) : TT t
| TyElim t : TT t
| TyAll t (ttt : TT t) (f : t → Type) (ftt : ∀ x, TT (f x))
  : TT (∀  (x:t), f x).

Arguments TyB _ {_}.
Arguments TyAll {_} _ {_} _.

Notation "! X" :=(TyB X) (at level 69).
Notation "X ⇝Y" :=(TyAll X (fun _ ⇒Y)) (right associativity, at level 70).
```

We allow the representation of a base type only if it can be encoded already. We mark the parts of a type which correspond to previously eliminated parts by introducing the constructor `TyElim`.

The `TyAll` constructor is able to represent dependent types. If we assume that we registered how to encode the type `sumbool`, the `TT` representation for `∀ x y : ℕ, { x = y } + { x ≠ y }` is

```
TyAll (! ℕ)
      (fun x : ℕ⇒
       TyAll (! ℕ) (fun y : ℕ⇒! {x = y} + {x ≠ y}))
  : TT (∀  x y : ℕ, {x = y} + {x ≠ y})
```

We define the `internalizesF` function by a proof script:

```
Definition internalizesF (p : Lvw.term) t (ty : TT t) (f : t) : Prop.
  revert p. induction ty as [ t H p | t H p | t ty internalizesHyp R ftt internalizesF'];
  simpl in *; intros.
  - exact (p >* enc f).
  - exact (p >* I).
  - exact (∀  (y : t) u, proc u → internalizesHyp y u → internalizesF' _ (f y) (app p u)).
Defined.
```

The idea is clear: `internalizesF (p : L.term) (t : Type) (ty : TT t) (f : t)` expresses the statement that the L-term p correctly internalizes the term f of type t. It works by structural recursion over the `TT` object. A term p internalizes $f : t$ for t being registered if $p \succ \overline{f}$. We require eliminated terms to reduce to the term I.

The function case where the term f has type $\forall t, R\,t$ is most interesting. We define that it is internalizes by a term p if $p\ u$ internalizes fy for every $y : t$ and internalization u of y that is a procedure. In the definition `internalizesF'` is essentially `internalizesF`, with some slight changes in the arguments.

```
Class internalizedClass (X : Type) (ty : TT X) (x : X) :=
{ internalizer : term ;
  proc_t : proc internalizer ;
  correct_t : internalizesF internalizer ty x
}.

Definition int (X : Type) (ty : TT X) (x : X) (H : internalizedClass ty x) :=internalizer.
Global Arguments int {X} {ty} x {H} : simpl never.

Definition correct (X : Type) (ty : TT X) (x : X) (H : internalizedClass ty x) :=correct_t.
Global Arguments correct {X} {ty} x {H} : simpl never.

Notation "'internalized' f" :=
  (internalizedClass $(let t :=type of f in let x :=toTT t in exact x)$ f)
  (at level 100, only parsing).
```

At first the decision to add a parameter instead of a new field seems odd. But, we can observe that there is at most one *meaningful* `TT` instance for every type. This statement is obviously (due to the term meaningful) nor provable in Coq, nor is the `TT` instance computable in Coq.

This is because the statement that there is at most one representation as `TT X` for every type `X` is true externally, but can not be proven in Coq. Not even a function `∀ X : Type, TT X` would be definable in Coq. This is why we need to add the `toTT` part as a parameter. If we assume that a function of type for instance `X → Y` is internalized, we also should be able to assume the right correctness lemma. Thus we add the `TT` object as parameter, but hide it for the user when writing it down.

4.2 Proving Correctness using Tactics

```
(* this can be used to internalize with an anonymous term, defined in coq*)
Ltac internalizeWith' t_arg:=
  match goal with
  | [ ⊢internalizedClass _ ?f ] ⇒
    let t :=(let t' :=eval lazy [enc] in t_arg in eval cbn in t') in
    let h :=visableHead t in
    (* this allows to branch if the head is opaque (e.g. a constructor) *)
    let e1 :=eval cbv in h in
    let e :=
        match h with
          e1 ⇒t
        | _ ⇒(let t' :=eval lazy [h enc] in t in t')
        end
    in
    let t' :=eval lazy [decision] in e in
    let ter :=to_iTerm_elim t' in
    let ter' :=annotateVariables ter in
    let Lter' :=toLambda ter' in
    match Lter' with
      | inl ?Lter
      ⇒apply Build_internalizedClass with (internalizer:=Lter); [try value|cbn]
    end
end.
```

The `internalizeWith'` tactic is the core of our tactics to assist the internalization of terms. It expects the term which should be internalized as argument `t_arg`.

If the head of the `t_arg` can be unfolded, it unfolds it. After that, every occurrence of `decision` is unfolded to its concrete instance. The resulting term gets converted to an `iTerm` and propositions and types get eliminated. At last, variables get annotated and the whole term is extracted. As a last step, the `Build_internalizedClass` constructor is applied.

There are two open goals: One is to prove that the internalized term is a procedure, which can be handled by the `value` tactic. The other goal is the correctness lemma itself, which is left open for the user.

With the help of the extensive automation from `Kunze15` these proofs can be done with few tactic calls in most cases. The only thing a user has to do is start the induction and do (if needed) some destructs by hand. Everything else, especially all reductions, is handled by the tactic `crush`.

5 Case Study

[2] develops a formalization of basic Computability theory in L. We reimplemented the formalization using the framework presented here. The following table presents a comparison in lines of Coq-code between the two approaches, concerning the verification of the most interesting procedures.

Formalization	**Thesis**	**Framework**
Natural Numbers	110	60
Equality on terms and $\mathbb{N}$	85	46
Lists	230	113
Substitution and Self Interpretation	209	74
Enumeration of terms	143	26
Inverse Encoding of $\mathbb{N}$	37	9
In Total	777	319

The definition of the framework and the corresponding tactics is roughly 650 lines long.

References

[1] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic: Volume II*. North-Holland Publishing Company, 1972.

[2] Yannick Forster. A formal and constructive theory of computation, Bachelor thesis, 2014. Available electronically at `https://www.ps.uni-saarland.de/~forster/bachelor.php`.

[3] Fabian Kunze. Bachelor thesis, work in progress, 2015. Available electronically at `https://www.ps.uni-saarland.de/~kunze/bachelor.php`.

[4] Gregory Malecha. *Template Coq*, 2015. Available electronically at `https://github.com/gmalecha/template-coq/`.

[5] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[6] Gert Smolka. Computation theory. Lecture Notes for Computational Logic 2, 2015. Available electronically at `https://courses.ps.uni-saarland.de/cl2/`.