A certifying extraction with time bounds from Coq to call-by-value λ-calculus

Yannick Forster
Saarland University, Saarland Informatics Campus (SIC), Saarbrücken, Germany
forster@ps.uni-saarland.de

Fabian Kunze
Saarland University, Saarland Informatics Campus (SIC), Saarbrücken, Germany
kunze@ps.uni-saarland.de

Abstract

We provide a plugin extracting Coq functions of simple polymorphic types to the (untyped) call-by-value λ-calculus L. The plugin is implemented in the MetaCoq framework and entirely written in Coq. We provide Ltac tactics to automatically verify the extracted terms w.r.t a logical relation connecting Coq functions with correct extractions and time bounds, essentially performing a certifying translation and running time validation. We provide three case studies: A universal L-term obtained as extraction from the Coq definition of a step-indexed self-interpreter for L, a many-reduction from solvability of Diophantine equations to the halting problem of L, and a polynomial-time simulation of Turing machines in L.

2012 ACM Subject Classification Theory of computation → Type theory; Mathematics of computing → Lambda calculus

Keywords and phrases cbv λ-calculus, Coq, constructive type theory, extraction, computability

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.10

Supplement Material The Coq development is accessible at
`https://github.com/uds-psl/certifying-extraction-with-time-bounds`

1 Introduction

Every function definable in constructive type theory is computable in a model of computation. This also enables many proof assistants based on constructive type theory to implement extraction into a "real" programming language. On the more foundational side, various realisability models for fragments of constructive type theory increase the trust in this meta-theorem, because realisers for types are the codes of computable functions.

The computability of all definable functions also enables the study of synthetic computability theory in constructive type theory [7, 4]. For instance, one can define decidability by `dec P := ∃ f, ∀ x, P x ↔ f x = true` and no reference to a concrete model of computation is needed. The undecidability of a predicate p can be shown by defining a many-one reduction from the halting problem of Turing machines to p in Coq, again without referring to a concrete model. The computability of all definable functions can, however, not be proved inside the type theory itself, similar to other true statements like parametricity. At the same time, for every concrete defined function of the type theory, one can always prove computability as theorem in the type theory. Given for instance any concrete function $f : \mathbb{N} \to \mathbb{N}$ definable in constructive type theory, one can construct a term of the λ-calculus t_f s.t. for all $n : \mathbb{N}$, there is a proof in the type theory that $t_f\ \overline{n}$ reduces to $\overline{fn}$ (where $\overline{\cdot}$ is a suitable encoding of natural numbers). The construction of t_f from f is relatively simple, since it is syntax-directed and the terms of type theory are just (possibly type-decorated) terms of an expressive untyped λ-calculus. Another way to see this construction is as extraction from the type theory into the λ-calculus.

We implement one such construction of λ-terms t_f for a certain subset of type theory: We use the MetaCoq framework [2] to extend the proof assistant Coq with a command to extract Coq functions of simple polymorphic types into the weak call-by-value λ-calculus L and provide tactics to automatically prove the correctness of the term. In addition to the correctness, our extraction command can generate recurrence equations that, if instantiated with a function by the user, describe the time complexity as number of β-steps of the extracted λ-term on its arguments. Our target calculus L has been used before to formalise computability theory in Coq [12]. Since it is (syntactically) the pure λ-calculus, recursive functions have to be encoded using a fixed-point combinator and inductive types using Scott's encoding.

Our extraction has several use cases:

First, while parts of computability theory can be formalised in Coq without referring to a model of computation [7], one needs a deep embedding of computable functions to e.g. construct universal machines. Our framework then allows the user to write all functions in Coq and automatically get λ-terms computing them, similar to practice on paper where function in the model are never spelled out. For instance, the automated construction of a universal λ-term takes about 30 lines and no manual proofs, whereas by hand construction and verification take about 500 lines [12].

Second, to the best of our knowledge, there are no formalisations of computational complexity theory in any proof assistant. We hope that our framework can be used to enable formalisations of basic complexity theory. One tedium – even on paper – when doing complexity theory in a way such that all details are spelled out is that constructing and verifying functions in the chosen model of computation is hard. With our framework, this burden is significantly lowered: Implementations can be given in Coq and only a suitable running-time function has to be given by hand. We extract a definition of Turing machines to show that L can simulate k steps of a Turing machine in a number of β-steps linear in k.

Third, synthetic undecidability and the notion of synthetic decidability and enumerability have been analysed in Coq [6, 7, 11, 20]. This resulted in a library of undecidable problems in Coq [10]. All problems of the library are shown undecidable by reduction from the halting problem of Turing machines. To show that all contained problems are actually interreducible with the halting problem, one has to give many-one reductions from the problems to the halting problem. Using extraction, a reduction to the halting problem for L is straightforward: It suffices to prove enumerability in Coq, which follows a clear scheme, and then extract the Coq enumerator automatically to L. We demonstrate the power of this method by reducing solvability of Diophantine equations to the halting problem of L.

Lastly, it might be beneficial to use classical axioms like choice when verifying reductions. Since the computability of all definable functions does not necessarily hold given classical assumptions, one can extract the used reductions to L to ensure their computability.

Related Work Myreen and Owens [25] implement a proof-producing translation from the higher-order logic implemented in the HOL4 system with a state-and-exception monad into CakeML [17]. The translation also produces proofs for the translated terms, similar to our approach. Hupel and Nipkow [14] give a verified compiler from a deep embedding of Isabelle/HOL to CakeML. Similar to our work, they use a logical relation to connect Isabelle definitions to an intermediate representation.

Mullen et al. [24] provide a verified compiler from a subset of Coq to assembly. Anand et al. [1] report on ongoing work on verifying the full extraction process of Coq, also based on the MetaCoq framework. They extract Coq functions into Clight, an intermediate language of the CompCert compiler, and are thus able to obtain verified assembly code for Coq functions.

Letouzey [21] describes the theoretical foundations of extraction in Coq. Our logical relation can be seen as a light-weight version of his simulation predicate for simple polymorphic types.

Köpp [16] verifies program extraction for functions in the Minlog proof assistant into a λ-calculus-like system.

Guéneau et al. [13] verify the asymptotic complexity of functional programs in Coq, based on separation logic with time credits.

We have reported on a preliminary version of our extraction plugin in [8].

2 The call-by-value λ-calculus L

We use the weak call-by-value λ-calculus L defined in [12] and based on [26, 19] as target language. It comes with an inductive type of terms

$$s, t, u, v \ : \ \mathbf{T} \ ::= \ n \mid st \mid \lambda s \qquad (n : \mathbb{N})$$

and a recursive function s_u^k providing a simple, capturing *substitution* operation:

$$\begin{aligned} k_u^k &:= u & n_u^k &:= n \quad (\text{if } n \neq k) \\ (st)_u^k &:= (s_u^k)(t_u^k) & (\lambda s)_u^k &:= \lambda(s_u^{1+k}) \end{aligned}$$

We will freely switch between a named representation for examples and the representation using de Bruijn indices for definitions, i.e. we write $\lambda xy.x$ for $\lambda\lambda 1$.

We define an inductive weak call-by-value *reduction relation* $s \succ t$:

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

We write $\succ^*$ for the reflexive transitive closure of $\succ$, $\succ^k$ for exactly and $\succ^{\leq k}$ for at most k steps.

Note that – contrary to Coq reduction – L-reduction does not apply below binders. Due to the capturing substitution relation, reduction is only well-behaved on closed terms. We call a term *closed* if it has no free variables. Closed abstractions are called *procedures* and are the (only) normal forms of normalising, closed terms.

L provides for recursion using a fixed-point operator:

Lemma 1 (Fact 6 [12]). *There is a function $\rho : \mathbf{T} \to \mathbf{T}$ s.t. $(\rho u)v$ reduces to $u(\rho u)v$ for procedures u, v.*

Inductive datatypes can be encoded using Scott encodings [23, 15], which we explain in Section 4.3.

One crucial property of L reduction is that it is uniformly confluent, making every reduction to a normal form have the same length:

▶ **Theorem 2** (Corollary 8 [12]). *If $s \succ^{k_1} v_1$, $s \succ^{k_2} v_2$ for procedures v_i, then $v_1 = v_2 \wedge k_1 = k_2$.*

For the remainder of this paper, we will write $\mathbb{T}$ for the type of types in Coq, $\mathbb{P}$ for the type of propositions, $\mathbb{L}\,X$ and $\mathbb{O}\,X$ for lists and options over X, and $\mathbb{1}$ (with $\star : \mathbb{1}$) for the unit type.

3 Correctness and time bounds

We define when a term computes a Coq function using two logical relations, one considering just correctness, and one correctness with time bounds. Crucial for both definitions is the notion of an encoding function:

Definition 3. *A function $\varepsilon_A : A \to \mathbf{T}$ is an encoding function for a type A if ε_A is injective and only returns procedures.*

Notice that the only types where such a function can be defined are computationally relevant (i.e. non-propositional), countable types like $\mathbb{B}$, $\mathbb{N}$, $\mathbb{O}\,X$, or $\mathbb{L}\,X$ over countable X.

3.1 Correctness

We define a logical relation $t_a \sim a$, meaning the L-term t_a correctly computes a. We will only define this predicate for elements $a : A$ where A is a simple type of the form $A_1 \to \cdots \to A_n$.

We define the predicate $t_a \sim a$ as follows:

$$\frac{}{\varepsilon_A a \sim a}\ (\text{for } a : A) \qquad \frac{\begin{array}{c} t_f \text{ is a procedure} \wedge \\ \forall a t_a.\ t_a \sim a \to \Sigma v : \mathbf{T}.\ t_f t_a \succ^* v \wedge v \sim fa \end{array}}{t_f \sim f}\ (\text{for } f : A \to B)$$

For elements $a : A$ for encodable types A the only term computing them is their encoding. Functions $f : A \to B$ are computed by a procedure t_f, if for every $a : A$ computed by t_a the term $t_f\ t_a$ computes fa. Note that we could alternatively define the first rule s.t. every term t convertible to the encoding $\varepsilon_A a$ computes a, and then simplify the second rule to read $t_f t_a \sim fa$. While technically correct, this simplification does not work for the extension of the relation with time complexity. We thus stick with the more complicated second rule where we require a term v (using the type theoretical sum Σ[1]) s.t. $t_f t_a$ reduces to v and v computes fa.

Defining this predicate in Coq is not entirely straightforward. As common when defining logical relations, the definition is not strictly positive and thus not accepted by Coq as inductive predicate. The standard approach for non strictly positive predicates is to translate them into a recursive function. However, here we would need recursion over types, which is not supported in Coq's type theory. We circumvent this restrictions by defining a type former $\mathfrak{T} : \mathbb{T} \to \mathbb{T}$ capturing exactly the types we want to recurse on and define the predicate by recursion on $\mathtt{ty} : \mathfrak{T} A$:

```
Inductive 𝔗 : Type → Type :=
  𝔗_base A `{registered A} : 𝔗 A (* base types *)
| 𝔗_arr A B (ty₁ : 𝔗 A) (ty₂ : 𝔗 B) : 𝔗 (A → B). (* functions types *)

Fixpoint computes {A} (ty : 𝔗 A) {struct ty}: A → T→ Type :=
  match ty with
    𝔗_base ⇒ fun x ext ⇒ (ext = enc x)
  | 𝔗_arr A B ty₁ ty₂ ⇒ fun f t_f ⇒ proc t_f * (* t_f is closed and normal *)
                                  ∀ (a : A) t_a, computes ty₁ a t_a →
                                  {v : term & (* there exists a term v*)
                                  (t_f t_a >* v)* computes ty₂ (f a) v} end.
```

The first constructor of $\mathfrak{T}$ takes every encodable type as argument, denoted in Coq by the `registered` type class, which we explain in Section 4.4. The second constructor captures exactly non-dependent functions. The definition of `computes` then exactly captures the inductive rules given above[2]. By making $\mathfrak{T}$ a type class, instances `ty` can always be obtained automatically.

As a running example, we will use the function $\mathtt{map}\ X\ Y : (X \to Y) \to \mathbb{L}\,X \to \mathbb{L}\,Y$ on lists for fixed types X and Y. We assume that X, Y, $\mathbb{L}\,X$ and $\mathbb{L}\,Y$ are all encodable. Then

[1] for non type-theoriest, Σ can be read as a computable existential quantifier
[2] Note that `{v : term & P v}` is Coq-notation for a dependent pair.

$t \sim \texttt{map}\ X\ Y$ is equivalent to t being a procedure and the proposition $\forall(f : X \to Y)(t_f : \mathbf{T})(L : \mathbb{L}\,X).\ t_f \sim f \to t\ t_f\ (\varepsilon L) \succ^* \varepsilon(\texttt{map}\ X\ Y f\ L)$.

Note that $\sim$ is defined similarly to $[\![\cdot]\!]_2$ on inductives and functions in [21].

3.2 Time bounds

We extend the computability predicate to include time bounds. As time measure for a term we use its number of β-steps to a normal form, which is shown reasonable in [9]. The time bound is expressed depending on the input itself, not its size: e.g. for $f : \mathbb{L}\,\mathbb{N} \to \mathbb{B}$ with $t_f \sim f$, we want to have a time complexity function $\tau_f : \mathbb{L}\,\mathbb{N} \to \mathbb{N}$ such that $\forall L : \mathbb{L}\,\mathbb{N}.\ t_f(\varepsilon L) \succ^{\leq(\tau_f L)} \varepsilon(fL)$.

We generalise this idea to also account for higher-order functions and define the type $\mathcal{C}$ of complexity measures τ_a for $a : A$ as follows:

$$\mathcal{C}A := \mathbb{1} \qquad \mathcal{C}(A \to B) := A \to \mathcal{C}A \to \mathbb{N} \times \mathcal{C}B$$

Given the term $\texttt{map}\ X\ Y$ of type $(X \to Y) \to \mathbb{L}\,X \to \mathbb{L}\,Y$ as above, its complexity measure $\tau_{\texttt{map}\ X\ Y}$ will be $(X \to Y) \to (X \to \mathbb{1} \to \mathbb{N} \times \mathbb{1}) \to \mathbb{N} \times (\mathbb{L}\,X \to \mathbb{1} \to \mathbb{N} \times \mathbb{1})$, which is equivalent to $(X \to Y) \to (X \to \mathbb{N}) \to \mathbb{N} \times (\mathbb{L}\,X \to \mathbb{N})$, i.e. it is a function that, given an argument $f : X \to Y$ and a complexity measure $\tau_f : \mathcal{C}(X \to Y)$ (being equivalent to $X \to \mathbb{N}$), returns a pair of the number of steps $\texttt{map}\,f$ needs to (partially) evaluate, and a function that for $L : \mathbb{L}\,X$ computes the remaining number of steps $\texttt{map}\,f\,L$ needs to evaluate.

We can extend the computability predicate with time bounds into a predicate $t_a \sim^{\tau_a} a$:

$$\frac{}{\varepsilon_A a \sim^{\tau} a}\ (\text{for } a : A) \qquad \frac{\begin{array}{c} t_f \text{ is a procedure} \wedge \\ \forall a t_a \tau_a.\ t_a \sim^{\tau_a} a \to \Sigma v : \mathbf{T}. \\ t_f t_a \succ^{\leq n} v \wedge v \sim^{\tau} fa \text{ where } \tau_f a \tau_a = (n, \tau) \end{array}}{t_f \sim^{\tau_a} f}\ (\text{for } f : A \to B)$$

The first rule is essentially unchanged: Since encoded terms $\varepsilon_A a$ are always normal, $\varepsilon_A a \sim^{\tau} a$ holds for every complexity measure τ. For the second rule, we decompose $\tau_f a \tau_a$ into n and τ. The complexity measure $\tau : \mathcal{C}B$ is the complexity measure for $v \sim^{\tau} fa$ and n is the number of steps $t_f\ t_a$ needs to reach v.

Similar to before, we implement the predicate by recursion on an element of $\mathfrak{T}A$:

```
Fixpoint computesTime {A} (ty : 𝔗 A) {struct ty}: A → T→ C A → Type := (* ... *).
```

4 Extraction

We describe the different tools needed to extract functions, constructors and to generate encoding functions.

4.1 Template-Coq

Template-Coq is a quoting library for Coq, now part of the MetaCoq project and originally developed by Malecha [22]. The current state of the project is explained by Anand et al. [2] and Boulier [5].

Template-Coq provides an inductive type `term` implementing the abstract syntax of Coq as an inductive type (Figure 1a). It comes with a monad `TemplateMonad : Type → Prop` (Figure 1b) which allows operations like quoting (i.e. converting Coq terms into their abstract syntax tree), unquoting (i.e. converting abstract syntax trees into Coq terms), evaluating terms, and making definitions. An operation `m : TemplateMonad A` can be executed using the `Run TemplateProgram m` vernacular command.

As an example, the following function obtains the type of its input by unquoting it into a pair of a type and an element, projecting out the type and returning its quotation:

```
Definition tmTypeOf (s : term) :=
  u ← tmUnquote s ;;
  u'←  tmEval hnf (my_projT1 u) ;;
  t ← tmQuote u';;
  ret t
```

```
Inductive term : Set :=
| tRel      : nat → term
| tLambda   : name → term (* the type *) → term → term
| tLetIn    : name → term (* the term *) → term (* the type *) → term → term
| tApp      : term → list term → term
| tConst    : kername → universe_instance → term
| tConstruct : inductive → nat → universe_instance → term
| tCase     : (inductive * nat) (* num of parameters *) →
              term (* type info *) → term (* discriminee *) →
              list (nat * term) (* branches *) → term
| tFix      : term → nat → term
(* ... *).
```

(a) Term representation

```
Inductive TemplateMonad : Type → Prop :=
(* Monadic operations *)
| tmReturn : ∀ {A:Type}, A → TemplateMonad A
| tmBind : ∀ {A B : Type}, TemplateMonad A →
             (A → TemplateMonad B) → TemplateMonad B

(* General commands *)
| tmPrint : ∀ {A:Type}, A → TemplateMonad unit
| tmFail : ∀ {A:Type}, string → TemplateMonad A
| tmEval : reductionStrategy → ∀ {A:Type}, A → TemplateMonad A

(* Return the defined constant *)
| tmDefinitionRed : ident → option reductionStrategy → ∀ {A:Type}, A → TemplateMonad A
| tmLemmaRed : ident → option reductionStrategy → ∀ A, TemplateMonad A

(* Quoting and unquoting commands *)
| tmQuote : ∀ {A:Type}, A → TemplateMonad term
| tmUnquote : term → TemplateMonad {T : Type & T}
| tmUnquoteTyped : ∀ A, term → TemplateMonad A
```

(b) Monad operations

Figure 1 Template-Coq's definitions

4.2 Extracting Terms

We define a monadic function `extract` which can extract admissible Coq terms into L. In order to extract a Coq term, all the constants appearing in it have to be extracted. To save work, we remember previously generated extracts, similar to Anand et al. [2], who use explicit dictionaries for this task. We employ Coq's type class mechanism instead of dictionaries:

```
Class extracted {A : Type} (a : A) := int_ext : T.
```

This also defines a function `int_ext` which allows referring to the extracted term corresponding to `a` as `int_ext a`, if it exists, and otherwise get an error.

```
Definition map (A B : Type) : (A → B) → list A → list B := fun f ⇒
  fix map := match l with | [] ⇒ @nil B | a :: t ⇒ @cons B (f a) (map l) end
```

Figure 2 Definition of `map : ∀ A B : Type, (A → B)→ list A → list B`

We restrict the terms we can extract to admissible terms:

▶ **Definition 4.** *A type A is admissible if A is of the form $\forall X_1 \dots X_n : \mathbb{T}.\ B_1 \to \dots \to B_m$ with $B_m \neq \mathbb{T}$. Terms $a : A$ are admissible if A is admissible and if all constants $c : C$ that are proper subterms of a are either*

1. *admissable and occur syntactically on the left hand side of an application fully instantiating the type-parameters of c with constants or*
2. *of type $\mathbb{T}$ and occur syntactically on the right hand side of an application instantiating type parameters.*

This means a type A is admissible if it has no quantification over terms, quantification over types in A is in prenex normal form and the return type of A is not $\mathbb{T}$. The function `map` (Figure 2) for instance is admissible. The only constants appearing in its body are `nil` and `cons`, which are both admissible and occur fully instantiated.

We define an extraction function which correctly extracts admissible terms of a type without type-parameters. If we want to extract polymorphic functions like `map` we use Coq's section mechanism and fix the types `A` and `B` as section variables and extract `map A B`.

The type of the extraction function is

```
extract : (nat → nat)→ term → nat → TemplateMonad T
```

The first argument is an environment argument which tracks lifting information for de Bruijn indices for the treatment of fixed points. The last argument is a fuel argument, needed because recursion on the right-hand constituents of an application is not structurally recursive.

Dealing with variables and binders is relatively straightforward, since Template-Coq already uses a de Bruijn representation of terms. Variables translate directly to variables, functions to λ and fixed points can be translated using ρ from 1. We have to lift variables when entering an abstraction using the standard de Bruijn lifiting operation ($\uparrow$):

```
Notation "↑ E" := (fun n ⇒ match n with 0 ⇒ 0 | S n ⇒ S (E n) end).

Fixpoint extract env s fuel :=
  match fuel with 0 ⇒ tmFail "out of fuel" | S fuel ⇒
  match s with
    Ast.tRel n ⇒ t ← tmEval cbv (var (env n));; ret t
  | Ast.tLambda _ _ s ⇒ t ← extract (↑ env) s fuel ;; ret (lam t)
  | Ast.tFix [BasicAst.mkdef _ nm ty s _] _ ⇒
      t ← extract (fun n ⇒ S (env n)) (Ast.tLambda nm ty s) fuel ;; ret (rho t)
```

In order to extract applications `s R` (where `R` is a list of all arguments), we count the number of type parameters of `s`. If it has none, extraction is straightforward recursion. We extract `s R` by folding over the list `R` as the application of the extraction of all subterms:

```
  | Ast.tApp s R ⇒
    p ← tmDependentArgs s;;
    if p =? 0 then
      t ← extract env s fuel;;
      monad_fold_left (fun t1 s2 ⇒ t2 ← extract env s2 fuel ;; ret (app t1 t2)) R t
```

If `s` has `p > 0` type parameters, we assume that it is the syntax of a previously extracted constant. We split `R` into type parameters `P` and the list of computational arguments `L` and unquote `tApp s P` as `a`. We then obtain an extraction `t` for the constant `a` using the `tmTryInfer` operation invoking type class search. Finally, we again recursively extract by folding over the list of arguments `L`:

```
    else
      let (P, L) := (firstn p R, skipn p R) in
      s' ← tmEval cbv (Ast.tApp s P);;
      (if closedn 0 s'
         then ret tt
         else tmFail "The term contains variables as type parameters.");;
      a ← tmUnquote s' ;;
      a' ← tmEval cbn (my_projT2 a);;
      n ← (tmEval cbv (String.append (name_of s) "_term") >>=tmFreshName) ;;
      i ← tmTryInfer n (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
```

For all other syntactic constructs we refer to the Coq code.

We wrap the extraction function into an operation which adds definitions:

```
Definition tmExtract (nm : option string) {A} (a : A) : TemplateMonad T:=
  q ← tmUnfoldTerm a ;;
  t ← extract (fun x ⇒ x) q FUEL ;;
  match nm with
    Some nm ⇒ nm ← tmFreshName nm ;;
                @tmDefinitionRed nm None (extracted a) t ;;
                tmExistingInstance nm;;ret t
  | None ⇒ ret t
  end.
```

4.3 Generation of Scott encodings

We use Scott encodings [23, 15] to encode inductive types and its constructors. Scott encodings represent the matches on the inductive type. For instance, the Scott encoding of the booleans are $\varepsilon_{\mathbb{B}}\texttt{true} = \lambda xy.x$ and $\varepsilon_{\mathbb{B}}\texttt{false} = \lambda xy.y$. For natural numbers, the encodings are $\varepsilon_{\mathbb{N}}0 = \lambda zs.z$ and $\varepsilon_{\mathbb{N}}(Sn) = \lambda zs.s(\varepsilon_{\mathbb{N}}n)$.

As before, we use type classes to remember previously generated encodings:

```
Class encodable (A : Type) := enc_f : A → T.
Class registered (A : Type) := mk_registered
  { enc :> encodable A ;          (* the encoding function for A *)
    proc_enc : ∀ a, proc (enc a); (* encodings are procedures *)
    inj_enc : injective enc       (* encoding is injective *) }.
```

For an inductive type with n constructors, the constructor of index i which takes a arguments has Scott encoding `gen_constructor` $a\ n\ i := \lambda x_1 \dots x_a.\lambda y_1 \dots y_n.y_i x_1 \dots x_a$.

For natural numbers (a type with two constructors, i.e. $n = 2$), the constructor S (which has index $i = 1$ and takes one argument, i.e. $a = 1$) has encoding $\lambda x.\lambda y_1 y_2.y_2 x$ (or $\lambda\lambda\lambda(02)$).

We use `gen_constructor` to define a monadic operation `tmExtractConstr`. If we want to extract `map`, we first extract the two constants occurring in its definition (i.e. `nil` and `cons`) and then the actual function, always fully applied to their type parameters:

```
Section Fix_X_Y.
  Context { X Y : Set }. Context { encY : encodable Y }.

  Run TemplateProgram (tmExtractConstr "nil_term" (@nil X)).
  Run TemplateProgram (tmExtractConstr "cons_term" (@cons X)).
```

```
  Run TemplateProgram (tmExtract "map_term" (@map X Y)).
End Fix_X_Y.
```

4.4 Generation of Encoding Functions

We restrict our generation of encoding functions to simple inductive types of the form

```
  Inductive T (X1 ... Xp : Type) : Type :=
  (* ... *) | constr_i_T : A1 → ... → An → T X1 ... Xp | (* ... *).
```

where `Aj` for $1 \leq$ `j` $\leq$ `n` is either encodable or exactly `T X1 ... Xn`.

For a fully instantiated inductive type `B = T X1 ... Xp` with `n` constructors we define the encoding function $\varepsilon_{\texttt{B}}$ as follows:

```
fix f (b : B) := match b with
 | ... | constr_i_T (x1 : A1) ... (xn : An) ⇒ λy1 ... yp.yi (f1 x1 ) ... (fn xn) | ... end
```

where `fj` for $1 \leq$ `j` $\leq$ `n` is a recursive call `f` if `Aj = B`, or $\varepsilon_{\texttt{Aj}}$ otherwise. We implement a monadic function `tmEncode` which can be used like this:

```
Section Fix_X.
  Variable (X:Type). Context {intX : registered X}.
  Run TemplateProgram (tmEncode "list_enc" (list X)).
End Fix_X.
```

Note that in principle, more types are Scott-encodable, but we leave the automatic generation for those types to future work.

4.5 Extraction in Coq

To be able to connect extracts t_a to terms a using the predicates $t_a \sim a$ and $t_a \sim^{\tau_a} a$ we define two type classes: The class `computable` is parameterised over a and contains an extracted term $t_a : \mathbf{T}$ and a proof of $t_a \sim a$. The class `computableTime` is in addition parametrised over a time complexity function τ_a:

```
Class computable {A : Type} {ty : 𝔗 A} (a : A) : Type :=
  { ext :> extracted a;
    extCorrect : computes ty a ext }.

Class computableTime {A : Type} (ty : 𝔗 A) (a : A): Type :=
  { extT : extracted a; evalTime : 𝒞 A ;
    extTCorrect : computesTime ty a extT evalTime }.
```

This way, we can write `ext a` or `extT a` for previously extracted terms t_a. Note that since all relevant information can be obtained through the paramaters and the types of the fields, we can leave all instances of this classes opaque in Coq.

5 Automated Verification

We now give an overview over the set of tactics we provide in our framework. All tactics are written in Ltac only, but some of them use the monadic operations explained in the last section. We first explain the tactics to simplify L-terms. We then show how to register inductive datatypes to be used with the framework. Lastly, we explain how to prove the computability relation $t_a \sim a$ and infer recurrence equations for a time bound τ_a.

5.1 Symbolic Simplification for L

All tactics in this section are concerned with proving goals of the form "s is a procedure" or "s reduces to t", or transforming a goal like "s reduces to t" to "s' reduces to t" by simplifying s to s'. While all terms s we simplify will be closed, they might not be concrete terms, e.g. contain the encoding of an arbitrary natural number. The tactics will not unfold definitions.

`Lproc`: The tactic `Lproc` can prove that a term is closed, an abstraction or a procedure. It syntactically decomposes the term and uses a hint database for easier extensibility.

`Lbeta`: The tactic `Lbeta` simplifies L-terms by reducing all β-redices of the form $(\lambda s)t$ which are visible without unfolding definitions. It uses `Lproc` to show that t is a procedure and that folded definitions used in s are closed, thus left unchanged by the substitution. `Lbeta` is implemented by reflection, treating names as opaque and using closures to evaluate big terms more efficiently. It can keep track of the number of beta-reductions performed. For example, it simplifies the L-term $(\lambda xy.\, xyy)\, u\, v$ in 2 steps to $u\, v\, v$.

`Lrewrite`: The tactic `Lrewrite` simplifies terms by the use of a hint database with the same name, containing the correctness statements for previously extracted terms, and by the use of local assumptions, which are important for recursion. For efficiency reasons, it does not use Coq's built-in rewriting and instead traverses terms to find subterms where a hint from the database is applicable. For example, it simplifies the L-term $t_+(t_+(\varepsilon_{\mathbb{N}}\ x)(\varepsilon_{\mathbb{N}}\ 5))(\varepsilon_{\mathbb{N}}\ y)$ to $\varepsilon_{\mathbb{N}}(x+5+y)$. While traversing, `Lrewrite` replaces occurences of t_y with $y : Y$ of registered type by the trivial instance with extraction $\varepsilon_Y y$. This guarantees canonicity of instances of `computable` for registered types.

Additionally, `Lrewrite` simplifies $t_f\, t_x$ to t_{fx} for $x : X$ and $f : X \to Y$. The concrete instance of `computable`(fx) is constructed by combining the instances for f and x.

`Lsimpl`: The tactic `Lsimpl` repeatedly applies `Lbeta` and `Lrewrite` in alternation and can solve trivial goals by reflexivity.

Time bounds: All tactics can be used to analyse time bounds as well: `Lbeta`, `Lrewrite`, and `Lsimpl` transform goals of the form $s \succ^{?k} t$ to goals of the form $s' \succ^{?k} t$ for an s' with $s \succ^{k1} s'$, instantiating the existential variable `?k` with `k1 + ?k'`.

5.2 Registering Inductive Datatypes

To register an inductive datatype we provide the monadic operation `tmGenEncode : ident → Type → TemplateMonad unit`:

```
Run TemplateProgram (tmGenEncode "nat_enc" nat).
Hint Resolve nat_enc_correct : Lrewrite.
```

The operation generates the encoding function and three obligations, which are discharged automatically[3]. The first and second obligation regard procedureness and injectivity of the generated encoding function by tactics `register_proc` and `register_inj`.

The third obligation is saved as `nat_enc_correct` and is generated similarly to the encoding function. It states that the encoding behaves like Scott encoding and is also proven automatically, using the tactic `extract match`. In the case of natural numbers, it has the following type: `nat_enc_correct : ∀ (n:nat)(s t:term), proc s → proc t → enc n s t` $\succ^{\leq 2}$ `match n with 0 ⇒ s | S n' ⇒ t (enc n') end`. The lemma has to be registered in the hint database `Lrewrite` manually in order to be used by our tactics.

[3] Using `Global Obligation Tactic` of the `Program` mode shipped with Coq.

To work with an inductive type, a user also has to extract its constructors. The constant constructors (e.g. 0 for natural numbers) are trivially computable by their encoding:

```
Instance reg_is_ext ty (R : registered ty) (x : ty) : computable x.
Proof. ∃ (enc x). reflexivity. Defined.
```

A specific instance is only needed for the functional constructors of inductive data types:

```
Instance term_S : computable S. Proof. extract constructor. Qed.
```

The `extract constructor` tactic extracts constructors as described in Section 4.3 and show their correctness fully-automatically as described in the next section.

5.3 Automatically Proving Correctness

As an example[4], we take the boolean disjunction `orb x y := if x then true else y`. For the user, the extraction is fully automatic:

```
Instance term_orb : computable orb. Proof. extract. Qed.
```

The tactic `extract` first extracts the Coq term as described in Section 4.2. In this case, the result is $\lambda xy.x(\texttt{ext true})y$. The verification is then performed by iterating the tactic `cstep`, where in each step a goal is of the form $s \sim \texttt{f}$. The tactic `cstep` performs simplifications depending on the Coq term `f`.

Here, the initial proof goal reads as follows:

$(\lambda xy.x(\texttt{ext true})y) \sim (\texttt{fun x y} \Rightarrow \texttt{if x then true else y})$

In case the Coq term is of function type and not syntactically a `fix`, `cstep` uses the definition of $\sim$ on function types and assumes a boolean `x` computed by a term `ext x`. This yields as intermediate goal the existence of a procedure v with

$(\lambda xy.x(\texttt{ext true})y)(\texttt{ext x}) \succ^* v$ and $v \sim (\texttt{fun y} \Rightarrow \texttt{if x then true else y})$

Now `cstep` uses `Lsimpl` to derive v by simplifying the term $(\lambda xy.x(\texttt{ext true})y)(\texttt{ext x})$ to $\lambda y.(\texttt{ext x})(\texttt{ext true})y$, yielding the proof goal

$\lambda y.(\texttt{ext x})(\texttt{ext true})y \sim (\texttt{fun y} \Rightarrow \texttt{if x then true else y})$

The next call of `cstep` assumes a fixed boolean `y` and simplifies by `Lrewrite`:

$\texttt{if x then ext true else ext y} \sim \texttt{if x then true else y}$

In case the Coq term syntactically has a case distinction on top, `cstep` performs the same case distinction for the proof, here leaving the two goals $\texttt{ext true} \sim \texttt{true}$ and $\texttt{ext y} \sim \texttt{y}$. In both cases the Coq term is of registered type and the next call of `cstep` proves these goals using the definition of $\sim$.

5.3.1 Recursive Functions

Recall that recursive functions in Coq are defined via the `fix` (or `Fixpoint`) construct, which allows the application of recursive calls to 'smaller' arguments, where the notion 'smaller' is due to the guardedness checker of Coq. The tactic `cstep` proves the correctness using `fix` as well, with the same recursive calls as the extracted function. Therefore, the guardedness checker will accept the proof for exactly the same reasons it accepted the function definition[5].

[4] Available as an interactive example in `Tactics/ComputableDemo.v` as `Example correctness_example`

[5] The guardedness checker rejects some of our produced proofs when extracting functions not directly structurally recursive: This is due to the additional heuristics in the guardedness checker.

As an example[6], the extraction of `map A B` (see Figure 2) for registered types `A` and `B` is of shape $\lambda f.\rho\ v_1$ for a procedure v_1, where ρ is the fixed-point combinator from Lemma 1.

To verify this term, the proof goal is

$\lambda f.\rho\ v_1 \sim$ `fun f ⇒ fix map l := (...)`

The first call of `cstep` is as in Section 5.3 and yields the following goal, where v_2 is obtained by replacing the `L`-variable f with `ext f` for a fixed computable `f : A → B` in v_1:

$(\rho\ v_2) \sim$ `fix map l := (...)`

In case the Coq term is syntactically a `fix`, `cstep` uses the definition of $\sim$ on function types, but generalises the goal over all arguments of `fix` (in this case only `l`):

$\forall$`l`.$\Sigma v : \mathbf{T}.(\rho\ v_2)$`(ext l)` $\succ^* v \wedge v \sim$ `(fix map l := (...))(ext l)`

`cstep` now inserts a a `fix` into the proof term, obtaining an inductive hypothesis `IH` of the same type as the goal. For the proof term to type-check in the end, `IH` can only be used on arguments structurally smaller than `l`. To guarantee this, `cstep` always performs a case analysis on the recursive argument first, i.e. in this case on `l`, yielding two goals.

In both resulting cases, `cstep` calls `Lrewrite` which uses the inductive hypothesis `IH` to simplify all occurrences of $(\rho\ v_2)$`(ext l')` to `ext ((fix map l := (...))l')`. In both goals, `cstep` needs to obtain a procedure v with $(\rho\ v_2)$`(ext l)`, which is done using `Lsimpl`. For `l` $= []$, the goal is trivial because `ext []` $\sim []$. In the recursive case `l = x :: l'`, `Lsimpl` yields the trivial goal

```
ext (f x :: ((fix map l := (...))l')) ~ f x :: ((fix map l := (...))l')
```

5.3.2 Higher-Order Functions

Terms containing higher-order functions applied to arguments need a syntactic transformation to be supported by our framework. To verify the correctness of e.g. `map (fun x ⇒ x + y)l` as part of a bigger program, we essentially need to show

$t_{\texttt{map}}\,(\lambda x.t_{+}\ x\ y)\,(\varepsilon$`l`$) \sim$ `map (fun x ⇒ x + y)l`

To use the definition of $\sim$ for $t_{\texttt{map}}$, we would have to show $(\lambda x.t_{+}\ x\ y) \sim$ `(fun x ⇒ x + y)`. This introduces several difficulties, one is that the term might contain free variables that need to be beta abstracted, and another one occurs when time bound are of interest: Since our verification of time bounds is only semi-automatic and requires the user to instantiate the recurrences by hand, we would need to interrupt the proof here for a user to fill in the concrete time bounds for $(\lambda x.t_{+}\ x\ y)$.

We thus restrict the scope of the framework and only cover applications of higher order functions to arguments which syntactically are composed from previously extracted term by application (without the use of abstractions). In this case this would mean that one has to define a Coq term `f y := fun x ⇒ x + y`, which has to be extracted before `map (f y)l`.

5.4 Proving Time Bounds

All simplification tactics also keep track of the number of β-steps in reductions and can thus be used to infer recurrence equations a correct time complexity function has to satisfy. The only obligation left to the user when proving instances of `computableTime` is to provide a solution to this recurrence equations. As an example, we consider boolean disjunction again and want to find a time complexity function $\tau : \mathbb{B} \to \mathbb{1} \to \mathbb{N} \times (\mathbb{B} \to \mathbb{1} \to \mathbb{N} \times \mathbb{1})$:

```
Instance term_orb : computableTime orb τ.
```

[6] Available as an interactive example in `Tactics/ComputableDemo.v` as `Example correct_recursive`

```
Proof. extract.
```

This leaves the user with the recurrence equations $\pi_1(\tau x\star) \geq 1$ and $\pi_1(\pi_2(\tau x\star)y\star) \geq 3$, indicating that $t_{\texttt{orb}}$ needs one step to reduce to an abstraction if applied to an encoded boolean x and this abstraction needs 3 further steps to a value if applied to a boolean y. Thus, choosing τ as `fun _ _ ⇒ (1,fun _ _ ⇒ (3,tt))` works. We provide the tactic `solverec` which simplifies goals containing inequations and tries to show them using the `lia` tactic shipped with Coq. If proving the inequality needs further reasoning, the tactic presents the user with simplified goals.

The recurrence equations for the time bound are inferred incrementally by `cstep` using an existential variable. To prove `computableTime orb` τ, `cstep` first introduces an assumption `H : ?P` τ and opens a new goal `?P` τ. In each step, `cstep` performs the transformations described in Section 5.3 while keeping track of the number of steps, asserting that τ needs to be larger than the number of β-steps performed by instantiating `?P` further. For non-recursive functions, this will only produce lower bounds for components of τ, while for recursive correctness proof it produces inequalities that contain τ on both sides.

To find time bound functions interactively, we define the opaque polymorphic constant `cnst {X:Type} (x:X): nat := 0` which can be used as a place-holder for unknown constants. To find the time complexity for `map`[7] one would start with the following:

```
Lemma termT_map A B (Rx : registered A) (Ry: registered B):
    computableTime (@map A B) (fun f τ_f ⇒ (cnst "c",fun l _ ⇒ (cnst ("g",l),tt))).
Proof. extract. solverec.
```

This yields three conditions: `1 <= cnst "c"`, `7 <= cnst ("g", [])`, and `fst (τ_f a tt) + cnst ("g", l) + 11 <= cnst ("g", a :: l)`. Note that `cnst` allows us to keep track of the different arguments that the time bound is instantiated with later. As expected, the time bound of map must also sum up all the time bounds for calling `f` on all elements of the list, and indeed, `solverec` can show the lemma using this time bound:

```
fun f τ_f ⇒ (1, fun l _ ⇒ (fold_right (fun x res ⇒ π1 (τ_f x tt) + res + 11) 7 l, tt))
```

6 Case studies

We provide three case studies: A universal L-term obtained as extraction from the Coq definition of a step-indexed self-interpreter for L (in `Functions/Universal.v`), a many-one reduction from solvability of Diophantine equations to the halting problem of L (in `Reductions/H10.v`), and a linear simulation of Turing machines in L (in `TM/TMEncoding.v`).

6.1 Step-indexed L-interpreter

A step-indexed interpreter for L is a function `eva` : $\mathbb{N} \to \mathbf{T} \to \mathbb{O}\,\mathbf{T}$ s.t. for closed s we have $(\exists n.\ \texttt{eva}\ n\ s = \lfloor t \rfloor) \leftrightarrow (s \succ^* t \wedge t\ \textit{is a procedure})$. The function can be defined as follows [12]:

```
Fixpoint eva (n : nat) (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
```

[7] Available as an interactive example in `Tactics/ComputableDemo.v` as `comeUp_timebound`

```
                    | Some (lam s), Some t ⇒ eva n (subst s 0 t)
                    |   _ , _ ⇒ None
  end          end    end.
```

Here `subst s 0 t` denotes substitution, which uses `Nat.eqb` as boolean equality test on natural numbers. We extract all three functions in reverse order. To do so, we first need encodings for natural numbers and term constructors as shown in Section 5.2 and encodings for terms. We first generate the encoding function and register it:

```
Run TemplateProgram (tmGenEncode "term_enc" term).
Hint Resolve term_enc_correct : Lrewrite.
```

We can then extract the non-constant constructors, `Nat.eqb`, `subst`, and `eva`:

```
Instance term_var : computableTime var (fun n _ ⇒ (1, tt)).
Proof. extract constructor. solverec. Qed.
Instance term_app : computableTime app (fun s1 _ ⇒ (1, (fun s2 _ ⇒ (1, tt)))).
Proof. extract constructor. solverec. Qed.
Instance term_lam : computableTime lam (fun s _ ⇒ (1, tt)).
Proof. extract constructor. solverec. Qed.

Instance termT_nat_eqb :
  computableTime Nat.eqb (fun x _ ⇒ (5, (fun y _ ⇒ ((min x y) * 15 + 8, tt)))).
Proof. extract. solverec. Qed.

Instance term_substT :
  computableTime subst (fun s _ ⇒ (5, (fun n _ ⇒ (1, (fun t _ ⇒
                        (15 * n * size s + 43 * (size s) ^ 2 + 13, tt)))))).
Proof. extract. solverec. Qed.

Instance term_eva : computable eva.
Proof. extract. Qed.
```

Note that the implementation of `eva` is very naive and needs steps exponential in n, we thus omit its time complexity.[8] A more reasonable implementation could be obtained by extracting the heap-based abstract machine from [18] to L.

6.2 Diophantine equations

The problems contained in the library of undecidable problems in Coq [10] are proven undecidable by a chain of many-one reductions starting at the halting problem for Turing machines. As a matter of fact, all problems contained in the library so far are actually interreducible. An easy way to prove this is to reduce leafs in the reduction graph to the halting problem for L defined as $\mathcal{E}s := \exists v.(s \succ^* v \wedge v \textit{ is an abstraction})$ and then implement one general reduction from $\mathcal{E}$ to the halting problem of Turing machines.

As an example how to reduce problems to $\mathcal{E}$ we use our framework to reduce solvable Diophantine equations [20], i.e. Hilbert's tenth problem H10, to $\mathcal{E}$.

We first explain the general structure using mathematical notation. In [7], the authors define synthetic notions of decidability and enumerability. If this definitions are enriched with explicit computability assumptions, one obtains:

▶ **Definition 5.** *A predicate* $p : X \to \mathbb{P}$ *is* L*-decidable if there exists* a computable $f : X \to \mathbb{B}$ *s.t.* $\forall x.\ px \leftrightarrow fx = \mathsf{tt}$.

[8] The recurrence equation generated for eva one would have to solve reads $f(1+n)(s_1 s_2) \geq f\ n\ s_1 + f\ n\ s_2 + 43 \cdot (\mathtt{size}\ t_1)^2 + f\ n\ (t_1{}^0_{t_2}) + 53$, with `eva` $n\ s_1 = \lambda t_1$ and `eva` $n\ s_2 = t_2$.

▶ **Definition 6.** *A predicate* $p : X \to \mathbb{P}$ *is* L*-enumerable if there exists* a computable $f : \mathbb{N} \to \mathbb{O}\,X$ *s.t.* $\forall x.\ px \leftrightarrow \exists n. fn = \lfloor x \rfloor$.

Theorem 7. *If* $p : X \to \mathbb{P}$ *is* L*-enumerable and equality on* X *is* L*-decidable, then* $p \preceq \mathcal{E}$.

Proof. Let f be the (computable) enumerator $\mathbb{N} \to \mathbb{O}\,X$ and $d : X \times X \to \mathbb{B}$ the (computable) equality decider. We define $s := \lambda x.\ \mu(\lambda n.\ t_f n\ (\lambda y. t_d\ x\ y)\ t_{\mathsf{ff}})$. Here, μ is an unbounded search operator, i.e. s performs unbouded search for x in the range of f. Then $px \leftrightarrow \mathcal{E}(s\ \overline{x})$. ◀

Moreover, it is easier to implement concrete enumerators based on lists, i.e. computable enumerators $f : \mathbb{N} \to \mathbb{L}\,X$ s.t. $px \leftrightarrow \exists n.\ x \in fn$. The equivalence proof of both notions can be found in [7]. Extending the proof with explicit computability assumptions as needed here is straightforward and we refer to the Coq code.

We now switch to a more technical notation and show how to construct such a list enumerator for H10 in Coq. We first define the type of polynomials, generate its encoding and extract its constructors:

```
Inductive poly : Set :=
    poly_cst : nat → poly          | poly_var : nat → poly
  | poly_add : poly → poly → poly | poly_mul : poly → poly → poly.

Run TemplateProgram (tmGenEncode "enc_poly" poly).
Hint Resolve enc_poly_correct : Lrewrite.

Instance term_poly_cst : computable poly_cst. extract constructor. Qed.
Instance term_poly_var : computable poly_var. extract constructor. Qed.
Instance term_poly_add : computable poly_add. extract constructor. Qed.
Instance term_poly_mul : computable poly_mul. extract constructor. Qed.
```

We define evaluation of polynomials under assignments `S : list nat` as and the decision problem `H10` as follows:

```
Fixpoint eval (p : poly) (S : list nat) :=
  match p with
  | poly_cst n ⇒ n
  | poly_var n ⇒ nth n S 0
  | poly_add p1 p2 ⇒ eval p1 S + eval p2 S
  | poly_mul p1 p2 ⇒ eval p1 S * eval p2 S
  end.
Definition H10 '(p1, p2) := ∃ S, eval p1 S = eval p2 S.
Instance term_eval : computable eval. extract. Qed.
```

where `nth n S d` returns the `n`-th element in `S`, or `d` if `S` is not long enough. We also define a computable function `poly_eqb : poly → poly → bool` deciding syntactic equality.

To show that H10 is L-enumerable, we enumerate all polynomials using `L_poly : nat → list poly`. Due to the restriction that higher-order arguments can not syntactically contain abstractions, we first extract uncurried versions of the constructors:

```
Definition poly_add' '(x,y) : poly := poly_add x y.
Instance term_poly_add' : computable poly_add'. extract. Qed.

Definition poly_mul' '(x,y) : poly := poly_mul x y.
Instance term_poly_mul' : computable poly_mul'. extract. Qed.

Fixpoint L_poly n : list (poly) :=
  match n with
  | 0 ⇒ []
  | S n ⇒ L_poly n ++ map poly_cst (L_nat n) ++ map poly_var (L_nat n)
                   ++ map poly_add' (list_prod (L_poly n) (L_poly n))
                   ++ map poly_mul' (list_prod (L_poly n) (L_poly n))
  end.
```

```
Instance term_L_poly : computable L_poly. extract. Qed.
```

The last and crucial lemma is the adaption of Fact 2.9 from [7]:

Lemma 8. *If $p : X \times Y \to \mathbb{P}$ is* L*-enumerable, then $\lambda x.\exists y.\ p(x, y)$ is* L*-enumerable.*

Theorem 9. H10 *is* L*-enumerable.*

Proof. By Lemma 8 we have to give a list enumerator for two polynomials `p1` and `p2` together with solutions `S`:

```
fix f n := match n with 0 ⇒ []
  | S n ⇒ f n ++ filter (fun '(p1,p2,S) ⇒ Nat.eqb (eval p1 S) (eval p2 S))
                  (list_prod (list_prod (L_poly n) (L_poly n)) (L_list_nat n)) end.
```

where `list_prod` is the cartesian product on lists and `L_list_nat` is a list enumerator for `list nat`. ◀

Corollary 10. H10 $\preceq \mathcal{E}$

Proof. By Theorems 9 and 7. ◀

6.3 Turing Machines

We show how our framework can be used to reduce the halting problem of multi-tape Turing machines `Halt` to the halting problem of L. We employ a Coq implementation of the definition of Turing machines by Asperti and Ricciotti [3], who formalise Turing machines in Matita.

```
Definition loopM : ∀ (sig : finType) (n : nat) (M : mTM sig n),
   mconfig sig (states M) n → nat → option (mconfig sig (states M) n) := (* ... *)

Definition Halt :{ '(Sigma, n) : _ & mTM Sigma n & tapes Sigma n} → _ :=
  fun '(existT2 _ _ (Sigma, n) M tp) ⇒
    ∃ (f: mconfig _ (states M) _), halt (cstate f) = true
                                 ∧ ∃ k, loopM (mk_mconfig (start M) tp) k = Some f.
```

Their formalisation uses the (dependent) vector type to model multiple tapes and an explicit transition function. Both aspects do not fit in our framework directly. We thus showcase two techniques to extend our framework in certain cases.

First, to encode types not in the scope of the framework, we notice that an encoding for a type A can be obtained from an encoding function ε_B given an injective function $A \to B$. We pack this insight in the definition `registerAs`, which can be used as follows:

```
Instance register_vector X `{registered X} n : registered (Vector.t X n).
Proof. apply (registerAs VectorDef.to_list). (* injectivity proof *) Defined.
```

Second, we observe that computability is closed under extensional equality:

Definition 11. *We define extensional equality for a type A with $\mathtt{ty} : \mathfrak{T}A$ recursively on $\mathtt{ty}$. Elements x, y of an encodable type A are extensionally equal if they are equal. Functions $f, g : A \to B$ are extensionally equal if for all $a : A$, fa is extensionally equal to ga.*

Lemma 12. *If f and g are extensionally equal and $t \sim^\tau f$ then $t \sim^\tau g$.*

Combining those two insights allows us to extract any vector operation by extracting the corresponding list-operation.

Furthermore, we use the fact that functions with finite domain and co-domain can always be translated into a value table containing lists of pairs. We can thus show that every transition function is computable in time independent of the current configuration, and derive time bound for `loopM`, executing a machine for k steps:

```
Instance term_trans : computableTime (trans (m:=M)) (fun _ _ ⇒ (transTime,tt)).
Proof. (* ... *) Qed.

Instance term_loopM :
  let c1 := (haltTime + n*121 + transTime + 76) in let c2 := 13 + haltTime in
  computableTime (loopM (M:=M)) (fun _ _ ⇒ (5,fun k _ ⇒ (c1 * k + c2,tt))).
Proof. unfold loopM. extract. solverec. Qed.
```

Here `haltTime` and `transTime` are constants depending on the concrete machine, its number of tapes and its alphabet. By unbounded search over all number of steps k we obtain:

Theorem 13. `Halt` *reduces to* $\mathcal{E}$.

7 Conclusion

Formalisation The tools in our framework heavily rely on Coq's tactic language Ltac to verify the correctness of extracted terms. During the verification, existential variables are crucial to generate the recurrence equations described in Section 5.4 while simultaneously simplifying the L-terms as described in Section 5.1. For this simplification, we implement a reflective simplification tactic for L-terms used in `Lbeta`. We tried to use setoid-rewriting for `Lrewrite`, but the need to track the number of reduction steps requires us to implement our own, domain-specific rewriting tactic in Ltac. This tactic implements bottom-up rewriting, resulting in smaller proof terms and faster rewriting, by performing many rewrite steps in one pass through the term: A tactic using congruence lemmas descends in the term and on the way out, rewriting steps are performed. We use the hint databases for the `auto`-tactic to add new lemmas for rewriting.

Typeclasses are employed as a kind of dictionary, e.g. to look up the extraction for a previously extracted function or its correctness lemma.

The framework consists of roughly 2100 lines of code, of which 370 are for the definitions described in Section 3.2 and their properties, 380 are for the extraction in Section 4, 950 are for the simplification presented in Section 5.1, and 420 are for the tactics proving those extracts correct in Section 5.3.

In total, the case studies consist of 340 lines of specification and 280 lines f code: 20 lines are for the universal machine, 200 for H10 and 400 for the Turing machine interpreter. All examples are built on a library of extracted functions concerning natural numbers, booleans and lists, which consists of 360 lines of code.

Future Work There are several directions in which the framework can be extended. We would like to extend the framework to support space bounds in addition to time bounds, based on the space measure defined in [9]. Furthermore, our automation framework is sound by construction, because it produces proofs. We conjecture it to be complete for the described fragment of Coq's type theory we are considering, but reasoning about tactics programmed in Ltac is basically impossible. In the future, we would like to be able to support all of Coq's type theory (possibly leaving out co-inductive types). In order to do that, the extraction process would have to support proof and type erasure, which can be implemented using Template-Coq.

On the more conceptual side, our extraction basically returns realisers in a realisability model for the treated fragment of Coq's type theory. We would like to analyse and verify such realisability models using MetaCoq, possibly connecting the (weak call-by-value) evaluation relation defined in MetaCoq with reduction in the realisability model, yielding a proof that for a certain subset of Coq's type theory, all definable functions are indeed computable.

Lastly, we hope that our framework enables the formalisation of basic computational complexity theory in Coq. We would like to mechanise results like a time hierarchy theorem for the call-by-value λ-calculus. The commonly known proofs for Turing machines or similar models use self-interpreters. The tightness of the provable gap then depends on the time-efficiency of the interpreter in use. As mentioned, the self-interpreter given in Section 6.1 is too inefficient and we want to extract the interpreters described in [18] and [9] to L.

References

1 Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

2 Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed Template-Coq. In *International Conference on Interactive Theorem Proving*, pages 20–39. Springer, 2018.

3 Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, October 2015. URL: `http://www.sciencedirect.com/science/article/pii/S0304397515006349`, `doi:10.1016/j.tcs.2015.07.013`.

4 Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006.

5 Simon Pierre Boulier. *Extending type theory with syntactic models*. PhD thesis, Ecole nationale supérieure Mines-Télécom Atlantique, 2018.

6 Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-related computational reductions in Coq. *arXiv preprint arXiv:1711.07023*, 2017. Accepted at ITP 2018.

7 Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 38–51. ACM, 2019.

8 Yannick Forster and Fabian Kunze. Verified extraction from Coq to a lambda-calculus. *Coq Workshop 2016*, 2016.

9 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *CoRR*, abs/1902.07515, 2019. `arXiv:1902.07515`.

10 Yannick Forster and Dominique Larchey-Wendling. Towards a library of formalised undecidable problems in Coq: The undecidability of intuitionistic linear logic. In *Workshop on Syntax and Semantics of Low-level Languages, Oxford*, 2018.

11 Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and minsky machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 104–117. ACM, 2019.

12 Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *ITP 2017*, pages 189–206. Springer, 2017.

13 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *European Symposium on Programming*, pages 533–560. Springer, 2018.

14 Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In *European Symposium on Programming*, pages 999–1026. Springer, 2018.

15 Jan Martin Jansen. Programming in the λ-calculus: From Church to Scott and back. In *The Beauty of Functional Code*, volume 8106 of *LNCS*, pages 168–180. Springer, 2013.

16 Nils Köpp. Automatically verified program extraction from proofs with applications to constructive analysis. Master's thesis, LMU Munich, 2018. URL: `http://www.mathematik.uni-muenchen.de/~schwicht/seminars/semws18/main.pdf`.

17 Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

18 Fabian Kunze, Gert Smolka, and Yannick Forster. Formal small-step verification of a call-by-value lambda calculus machine. In *Asian Symposium on Programming Languages and Systems*, pages 264–283. Springer, 2018.

19 Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008. doi:10.1016/j.tcs.2008.01.044.

20 Dominique Larchey-Wendling and Yannick Forster. Hilbert's Tenth Problem in Coq. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:20, 2019.

21 Pierre Letouzey. *Certified functional programming: program extraction within Coq proof assistant*. PhD thesis, Université Paris-Sud, France, 2004.

22 Gregory Michael Malecha. *Extensible proof engineering in intensional type theory*. Harvard University, 2015.

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

24 Eric Mullen, Stuart Pernsteiner, James R Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 172–185. ACM, 2018.

25 Magnus O Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2-3):284–315, 2014.

26 Gordon D. Plotkin. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.