Proof Pearl: Faithful Computation and Extraction of μ-Recursive Algorithms in Coq

Dominique Larchey-Wendling
Université de Lorraine, CNRS, LORIA, Vandœuvre-lès-Nancy, France

Jean-François Monin
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France

Abstract

Basing on an original Coq implementation of unbounded linear search for *partially decidable* predicates, we study the computational contents of μ-recursive functions via their syntactic representation, and a correct by construction Coq interpreter for this abstract syntax. When this interpreter is extracted, we claim the resulting OCaml code to be the natural combination of the implementation of the μ-recursive schemes of composition, primitive recursion and unbounded minimization of *partial* (i.e., possibly non-terminating) functions. At the level of the fully specified Coq terms, this implies the representation of higher-order functions of which some of the arguments are themselves partial functions. We handle this issue using some techniques coming from the Braga method. Hence we get a faithful embedding of μ-recursive algorithms into Coq preserving not only their extensional meaning but also their intended computational behavior. We put a strong focus on the quality of the Coq artifact which is both self contained and with a line of code count of less than 1k in total.

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Type theory; Theory of computation → Functional constructs; Software and its engineering → Formal methods; Software and its engineering → Functional languages; Theory of computation → Higher order logic

Keywords and phrases Unbounded linear search, μ-recursive functions, computational contents, Coq, extraction, OCaml

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.21

Supplementary Material *Software*: `https://github.com/DmxLarchey/Murec_Extraction` archived at `swh:1:dir:4d128568b56a17277c4f69ee1805e3910665f34f`

Funding *Dominique Larchey-Wendling*: partially supported by NARCO (ANR-21-CE48-0011).

1 Introduction

The theory of μ-recursive functions is a well established and widely used model for representing (partial) recursive functions of type $\mathbb{N}^k \rightharpoonup \mathbb{N}$ where $\mathbb{N}^k$ is the type of tuples of natural numbers of arity k. It originates from primitive recursive functions, invented in the 1920s in the Hilbert school (the modern denomination was coined by Rózsa Péter), which is the smallest class of functions containing constant functions, the successor function, projections (of the i-th argument), and closed under the schemes of composition and primitive recursion. Primitive recursive schemes define provably total functional relations but do not cover all the spectrum of computability, the Ackermann function giving the most popular counter-example.

Gödel defined the larger class of "general" recursive functions, developing ideas of Herbrand, and Kleene [7] later proposed to augment the allowed primitive recursive schemes with that of unbounded minimization of partial functions, giving the class of μ-recursive functions, equivalent (extensionally) to that of general recursive functions, and of which primitive recursive functions form a natural, strict sub-class.

```
(** val ra_compute : recalg → nat list → nat **)
let rec ra_compute s v =
  match s with
  | Ra_zero → O
  | Ra_succ → match v with y :: _ → S y
  | Ra_proj i → vec_prj v i
  | Ra_comp (s, s_vec) → ra_compute s (vec_map_compute (fun t → ra_compute t v) s_vec)
  | Ra_prec (s, s'') → match v with y :: u →
    prim_rec_compute (ra_compute s) (fun w n x → ra_compute s'' (n :: x :: w)) u y
  | Ra_umin s' → umin_compute (fun n → ra_compute s' (n :: v)) O
```

Figure 1 An OCaml interpreter for μ-algorithms.

In the context of mechanization, μ-recursive functions have been implemented and/or used in several projects [13, 9, 2, 10]. Most of these developments are concerned with computability theory (the S-n-m theorem, Rice's theorem, Kleene's normal form theorem, Hilbert's tenth problem) so they mainly focus on their extensional properties. On the contrary, in this proof pearl, we mostly focus on the intentional contents of μ-recursive schemes. We call these μ-algorithms and reserve the term "function" for the extensional notion.

Of course, μ-algorithms do not provide all the algorithmic means to compute values: for instance, they lack course-of-values recursion, nested recursion, higher-order primitive recursion, higher-order functions, and they are grounded on the very rudimentary datatype natural numbers, hence, when e.g. computing with lists or trees, one must pass through encoders and decoders. However, μ-recursive functions are universal in that they capture all computable functions and, in comparison with other models with the same power, they have the usual "referential transparency" advantage of functional languages over imperative ones, which *naturally* makes them better suited to equational and compositional reasoning than, say, Turing or Minsky (counter) machines.

In a contemporary approach, μ-algorithms can be described by an abstract syntax and a simple interpreter providing the computation rules to be followed by each construct. This abstract syntax can be implemented by the following type, where `Ra_comp` encodes composition, `Ra_prec` encodes primitive recursion, `Ra_umin` encodes minimization, and `Ra_zero`, `Ra_succ` and `Ra_proj` are obvious.

```
type nat = O | S of nat
type recalg = Ra_zero | Ra_succ | Ra_proj of nat | Ra_comp of recalg * recalg list |
              Ra_prec of recalg * recalg | Ra_umin of recalg
```

A natural OCaml program for interpreting pieces of code written in the above language is displayed in Figure 1. The arguments (s, s_{vec}) of `Ra_comp` stand respectively for (the source code of) a n-ary function and a n-tuple of functions, whose output is expected to be fed as inputs for (the interpreter of) s. The arguments (s, s'') of `Ra_prec` stand respectively for the result to be returned in the base case (the last argument is zero) and the function to be applied in the step case (the last argument is a successor). For convenience, the tuple $\langle x_1, \ldots, x_k \rangle$ of natural numbers representing the inputs of a μ-recursive algorithm is encoded by a list in the reverse order $[x_k ; \ldots ; x_1]$, so that the driving argument n for primitive recursion is the head of the list.

The functional implementations of μ-recursive schemes are straightforward: `vec_prj` for projections, `vec_map_compute` for compositions, `prim_rec_compute` for primitive recursion and `umin_compute` for μ-minimization.

```
(** val vec_prj : α list → nat → α **)
let rec vec_prj u i = match u with
  | x :: v → match i with O → x | S j → vec_prj v j

(** val vec_map_compute : (α → β) → α list → β list **)
let rec vec_map_compute f = function
| []     → []
| x :: v → f x :: vec_map_compute f v

(** val prim_rec_compute : (α → β) → (α → nat → β → β) → α → nat → β **)
let rec prim_rec_compute f g x = function
| O   → f x
| S n → g x n (prim_rec_compute f g x n)

(** val umin_compute : (nat → nat) → nat → nat **)
let rec umin_compute f n = match f n with
| O   → n
| S _ → umin_compute f (S n)
```

Though the above code is not very complicated, there are some subtle points making it a not-so-trivial case study if we want to prove its correctness w.r.t. a formal specification. In particular, as a purely functional piece of code, it seems reasonable to get it by extraction of a Coq proof. However, unbounded minimization or μ-minimization of a (partial) function $f : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ consists in the (partial) function $\mu f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that $\mu f(x)$ is the number n for which $f(n, x)$ is 0 and $f(i, x)$ is defined and not equal to 0 for every number i less than n. Although uniquely defined, it may not exist. For instance, μ-minimization of a constant (non-zero) function (e.g. $f(\cdot, \cdot) = 1$) is the nowhere defined function of type $\mathbb{N} \rightharpoonup \mathbb{N}$. Indeed, as soon as μ-minimization is available, *all* the other constructs are contaminated and encode possibly non-terminating functions. Altogether, not only `ra_compute` is clearly undefined on some inputs (where computation does not terminate), but it relies on higher-order programs such as `vec_map_compute` and `prim_rec_compute` which have to be themselves considered as non-terminating since they are applied to *possibly non-terminating arguments*, all in a nested recursive manner.

In a previous work [9], the first author showed that it is possible to derive a Coq term which computes the same result as a given μ-recursive function. The latter was first transformed, by bounding it using the folklore "fuel" technique, thus giving a primitive recursive (hence terminating) algorithm, and then applying Constructive Epsilon, i.e. unbounded minimization of inhabited and decidable predicates over $\mathbb{N}$. Kleene used a comparable trick in order to establish that every μ-recursive function can be obtained from the μ-minimization of some primitive recursive function, i.e. the normal form theorem [8]. In his section "resource bounded evaluation," Carneiro [2] uses the same fuel trick in Lean to approximate partial computable functions with primitive recursive ones, combined with a variant of Constructive Epsilon he calls find.

Though this approach is sufficient for theoretical purposes such as studying the expressive power of computational models,[1] it is unsatisfactory from an algorithmic point of view: the underlying calculation boils down to a systematic and heavy trial-and-error process that

[1] Most of the textbook presentations of μ-recursive functions like e.g. [12, 1] focus on their extensional meaning as set-theoretic partial functions, i.e., the relation between inputs and outputs, or so called graph, but not on the calculations performed.

is unfaithful to the intended behavior, unlike the intuitive OCaml code above. Directly reasoning on the latter with extraction in mind is actually more demanding: we need to express the computational counterpart of the desired specific program in a type-theoretic framework where only total functions are allowed. In other words, we need to provide "generic" partial termination certificates for (the Coq counterpart of) the above OCaml functions.

We show here how the Braga method [11], our trick to manage termination issues in recursive programs, can be adapted in the present case study in order to satisfy the above requirements. In a nutshell, the Braga method allows us to define functions of x whose termination is provided by an additional domain argument $d : D\,x$, of sort `Prop`, where D is inductively defined and systematically derived from the shape of the desired program `Pgm` to be extracted. In this way we can express a functional program

$$\texttt{Fixpoint}\ g\ x\ (d : D\,x)\ \{\texttt{struct}\ d\} :=\ \ldots\ g\,x'\,d'\ \ldots$$

where $d' : D\,x'$ is structurally smaller than d – the *guard condition* which ensures termination. Then we can reason on the partial correctness properties of g before (and without) bothering about termination (e.g., without defining a measure which, anyway, could not exist in all cases). The domain D is typically the domain of an inductive input-output graph G, which is nothing but a relational presentation of `Pgm` and can be seen as a complete characterization of `Pgm`. In [11], G is also used in the Σ-type of $g : \forall x,\ D\,x \to \{y \mid G\,x\,y\}$ so that, after erasure, the extracted OCaml program automatically satisfies this characterization as well as the partial correctness properties which are derived from it.

In the case of μ-algorithms we need to go further than [11]. First, partial functions are essential here whereas almost all examples considered in [11] turn out to be total functions. Second, we already have a natural input-output graph: a relational semantics for the abstract syntax of μ-algorithms. However, this semantics is naturally expressed as a combination of partial functions, one for each construct of μ-algorithms. In order to mimic the original formulation, we encode partial functions from $\mathbb{N}^k$ to $\mathbb{N}$ by total functions from $\mathbb{N}^k$ to $\mathbb{N} \to$ `Prop`, yielding a compact and crystal-clear specification. This semantics is detailed in section 4.2 and noted $[\![.]\!]$. Looking back at the intended program above, we need to extend the Braga method to get Coq programs that: first, combine in a similar and hopefully modular way an implementation of μ-minimization with "ordinary looking" structural recursion on data structures such as `nat` or the syntax of μ-algorithms; and second, are driven by the domain of an input-output graph G, such as $[\![.]\!]$, that is not necessarily expressed inductively. A suitable general type scheme for such programs is simply $\forall x,\ (\exists y, G\,x\,y) \to \{y \mid G\,x\,y\}$, which can be implemented either by ordinary structural recursion on the first input x, or by using the Braga method on a termination certificate derived from the second input $(\exists y, G\,x\,y)$.

Additionally, μ-minimization provides another opportunity to change the Braga machinery a little bit: for `umin_compute`, which is basically a tail-recursive presentation of an (unbounded) loop for linear search, the propagation of assertions is not encoded backwards by a Σ-type embedding of the expected input-output relation, but forwards by a parameter containing an inductive invariant. This makes the enriched Coq program already tail-recursive and, more importantly, proofs of propagation become simpler.

Remarks about the Coq code. The artifact we publicly deliver only requires a minimal amount of Coq machinery to implement linear search and the interpreter for μ-recursive algorithms. To illustrate this and also produce self-contained code, we do not use the Coq standard library, except for the modules `Utf8` to allow for better human readable Coq code, and `Extraction` to witness our claims about the faithfulness of the extracted code to the

"natural" OCaml interpreter. The code is intended to be read as an essential part of this pearl, and we invite the reader to consult the associated artifact, starting with the `README.md` file. We tried to make the artifact readable by a human, without the help of the type-checker. This means that proofs written with scripts (`Ltac`) are both very short, and with only light automation not beyond `trivial` or `easy`. When the contents of terms is critical, e.g. when it contributes to extraction, or in order to visualize structural decrease, we write these as λ-terms.

Contributions. We hope our contributions to be somewhat valuable for people using Coq or a similar Type Theory as a programming language, but nothing original is claimed about computability theory. First of all, we provide a short, clean, readable and (hopefully) informative, Coq implementation of the partial linear search algorithm, extending Constructive Epsilon to partially decidable predicates, using a variant of the Braga method that also fully takes into account the tail recursivity of the underlying program. Second, we contribute a Coq interpreter for μ-recursive algorithms, which follows their intended (functional) operational semantics, taking [1] as reference, as witnessed by a neat extraction to OCaml. Third, this interpreter relies on linear search in a way that illustrates a general approach to integrating programs written with the Braga method (or variants of it) and structural recursive programs that depend on each other in a nested or mutual recursive way. With this reasonably sized and documented code, we also hope to popularize further some dependent inversion techniques that sometimes hinder the usage of Coq structural fixpoints. For instance, see our small library for dependent vectors that features improvements on the standard library and is of independent interest.

This proof pearl is constructed as follows. Section 2 provides the prerequisites needed to understand the paper and the associated Coq artifact. Section 3 explains how to generalize the specification of the linear search algorithm to be able to search with partially decidable predicates. This generalization of Constructive Epsilon is the essential ingredient to implement the scheme of μ-minimization. Section 4 presents the Coq implementation of μ-algorithms with their extensional semantics, following [1], and then their intentional contents as a Coq term, an interpreter computing the output, if provided with a proof of termination on the given input. Section 5 presents the result of the extraction of the above mentioned interpreter as an OCaml program. We list the various tweaks that help at completely rendering a readable (and herein presentable) program. We also explain how to get rid of some possibly unwanted OCaml tricks used by the extraction plugin to circumvent the limitations of the OCaml type-system, as compared to that of Coq, the source type theory.

2 Type-theoretic basics and notations

We present this pearl in the language of the type theory of Coq containing the sort `Prop`, the impredicative type of propositions, and the sort `Set`, the predicative type at the ground-level of sorts in `Type`, the predicative type hierarchy above both `Prop` and `Set`.

The basic inductive structures we use are the propositions (`True : Prop := I`) and (`False : Prop :=` .), the data types `unit : Set := tt`, first-order logic connectives and Peano natural numbers `nat : Set := O | S (_ : nat)`, endowed with natural ($\leq$) and strict ($<$) orders. Only a minimal amount of basic arithmetic results are needed, for which we give tiny proofs in `arith_mini.v`. We won't use lists but vectors instead, see Section 4.4.

We will use tuples (n-ary products) and dependent pairs (Σ-types) that come under various forms in Coq, see files `sigma.v`, `relations.v` and `vec.v`. We will write e.g. $\langle a, b, c\rangle$ for a triple of three values that may be proofs of propositions, hence giving of a proof of say $A \wedge B \wedge C$, or else terms giving a value in the product type $A \times B \times C$.

Given a predicate $P : X \to \texttt{Prop}$, we write dependent pairs as $\langle\langle x, p\rangle\rangle$ where p is a proof of $P\,x$ (hence dependent on x). In this paper, depending on the context (`Prop` vs. `Type`), $\langle\langle x, p\rangle\rangle$ denotes either a proof of the proposition $\exists x, P\,x$: `Prop`, also written $\texttt{ex}\,P$, or an inhabitant of the type $\{x \mid P\,x\} : \texttt{Type}$, also written $\texttt{sig}\,P$. Likewise, given $d := \langle\langle x, p\rangle\rangle$, we write $\pi_1\,d := x$ and $\pi_2\,d := p$ for the projections of the dependent pair d.

In the file `relations.v`, we also give basic tools to manipulate 0-ary, unary and binary relations (i.e. predicates), like notations for composition, inclusion, conjunction. A unary relation $P : X \to \texttt{Prop}$ is *functional (or deterministic)* if there is at most one x s.t. $P\,x$.

3 Unbounded linear search in Coq

The linear search algorithm on an unbounded interval of $\mathbb{N}$ (LS for short) is the main engine of μ-minimization. As one of the simplest examples of a program which computes a *partial* (recursive) function, it is particularly interesting. It can be specified as follows. Given a decidable predicate P on $\mathbb{N}$ and a starting number $s : \mathbb{N}$, find an $n : \mathbb{N}$ greater or equal to s such that $P(n)$. Among its many other applications, we can cite Constructive Epsilon, which corresponds to the special case where $s = 0$. More precisely, it realizes the specification $\texttt{ex}\ P \to \texttt{sig}\ P$, a short hand for $(\exists x, P\ x) \to \{n \mid P\ n\}$. As we reuse some ideas coming from `ConstructiveEpsilon.v` of the Coq standard library, the name "Constructive Epsilon" will below refer to this implementation.

Assuming a suitable program `test`, here is an obvious OCaml program for unbounded linear search:

```
let rec loop s = if test s then s else loop (s + 1)
```

Let us informally write $\texttt{Gr\_than}_P(s)$ for the set of natural numbers x such that $x \geq s$ and $P\,x$ holds. If $\texttt{Gr\_than}_P(s)$ is inhabited, the returned value is actually the *least* value m in $\texttt{Gr\_than}_P(s)$; but otherwise, the algorithm loops forever. The underlying function is then clearly *partial*. In the following we discuss the contents of file `linear_search.v`.

3.1 Specification of linear search

Aiming first at a general Coq specification of the unbounded linear search algorithm, we actually don't need to assume that P is decidable on `nat`, but only between s and a large enough number. For additional generality and convenience we also consider two predicates $D_{\texttt{test}}$ (the domain of `test`) and Q such that, whenever $D_{\texttt{test}}$ holds, P or Q can be decided and P and Q cannot hold together.

$$\texttt{test} : \forall n,\ D_{\texttt{test}}\ n \to \{P\ n\} + \{Q\ n\} \qquad \texttt{PQ\_abs} : \forall n,\ D_{\texttt{test}}\ n \to P\ n \to Q\ n \to \texttt{False}$$

We then only assume that $D_{\texttt{test}}$ holds between s and a large enough number. On the side of the post-condition, we see that not only $D_{\texttt{test}}$ and P hold at the returned value m, if any, but also that $s \leq m$ and that $D_{\texttt{test}}$ and Q hold at all k such that $s \leq k < m$. Defining

$$\texttt{Definition btwn}\ (A : \texttt{nat} \to \texttt{Prop})\ n\ m := n \leq m \wedge (\forall k,\ n \leq k < m \to A\ k) \tag{1}$$

and with the notation $A \wedge_1 B := \lambda n,\ A\,n \wedge B\,n$, the strongest post-condition characterizing the output of LS is then:

$$\texttt{Definition}\ \mathbb{P}\texttt{ost}_{\texttt{ls}}\ s\ x := (D_{\texttt{test}}\ x \wedge P\ x) \wedge \texttt{btwn}\ (D_{\texttt{test}} \wedge_1 Q)\ s\ x.$$

On the side of the pre-condition of LS, we assume the existence of some x in $\texttt{Gr\_than}_P(s)$ and the ability to perform `test` from s to x. It can be stated using the following predicate.

```
Definition ℙre_ls s x := (D_test x ∧ P x) ∧ btwn D_test s x.
```

The expected type for linear search (starting at s) is then $(\exists x,\ \mathbb{P}\texttt{re}_{\texttt{ls}}\ s\ x) \to \{m \mid \mathbb{P}\texttt{ost}_{\texttt{ls}}\ s\ m\}$.

3.2 Termination of linear search

Observe that the witness x mentioned in the pre-condition is not used in the search loop – it is not available at the informative level. Moreover, `btwn` $Q\ s\ x$ is not assumed, that is, x is not necessarily minimal in $\texttt{Gr\_than}_P(s)$. But x can be used to compute a *termination certificate* since its very existence guarantees that the search loop eventually halts. The usual argument, in an imperative setting, consists in proving that $x - s$ is a loop variant. However, as mentioned in the introduction, we can take advantage of an essential feature of type theory of Coq to provide a direct inductive characterization of termination of sort `Prop` called $\mathbb{D}\texttt{ls}$, to be used as follows: `Fixpoint loop` $n\ (d : \mathbb{D}\texttt{ls}\ n)\ \{\texttt{struct}\ d\} := \ldots$ Only n can be used in the informative part of the computation; on the other hand, a strict sub-term of d has to be provided in any recursive call. Consistently, d is erased at extraction time. The design of $\mathbb{D}\texttt{ls}$ follows the pattern of recursive calls of the target program and keeps track of the information resulting from the tests carried out. Here is our definition of $\mathbb{D}\texttt{ls} : \texttt{nat} \to \texttt{Prop}$:

$$\frac{c : D_{\texttt{test}}\ n \wedge P\ n}{\mathbb{D}\texttt{ls\_stop}\ c : \mathbb{D}\texttt{ls}\ n} \qquad \frac{t : D_{\texttt{test}}\ n \qquad d_{\texttt{S}} : \mathbb{D}\texttt{ls}\,(\texttt{S}\ n)}{\mathbb{D}\texttt{ls\_next}\ t\ d_{\texttt{S}} : \mathbb{D}\texttt{ls}\ n}$$

Defining $\mathbb{D}\texttt{ls\_}\pi_1\ d$ as the immediate $D_{\texttt{test}}\,n$ component of $d : \mathbb{D}\texttt{ls}\,n$ using an easy pattern-matching, and $\mathbb{D}\texttt{ls\_}\pi_2\ d\ q$ as the immediate $\mathbb{D}\texttt{ls}\,(\texttt{S}\ n)$ component of d when $q : Q\ n$, using a more subtle pattern-matching, a version of `loop` returning a simple natural number is

```
Fixpoint loop n (d : 𝔻ls n) {struct d} : nat :=
  match test n (𝔻ls_π₁ d) with
  | left p  ⇒ n
  | right q ⇒ loop (S n) (𝔻ls_π₂ d q)
  end.
```

The second projection $\mathbb{D}\texttt{ls\_}\pi_2\,d$ returns a λ-term of type $Q\,n \to \mathbb{D}\texttt{ls}\,(\texttt{S}\ n)$ for each constructor. In the easy (and "intended") case where d is $\mathbb{D}\texttt{ls\_next}\ t\ d_{\texttt{S}}$, it just returns $\lambda\,\_,\ d_{\texttt{S}}$. And when d is $\mathbb{D}\texttt{ls\_stop}\ \langle t, p\rangle$, then t, p and q conspire with `PQ_abs` to construct a proof of the empty type `False`, on which an additional pattern matching provides a strict sub-term of any proof (of any inductive predicate).

```
Definition 𝔻ls_π₂ {n} (d : 𝔻ls n) : Q n → 𝔻ls (S n) :=
  match d with
  | 𝔻ls_stop ⟨t, p⟩ ⇒ λ q, match PQ_abs t p q with end
  | 𝔻ls_next _ d_S ⇒ λ _, d_S
  end.
```

The braces around the first parameter $\{n\}$ mark an *implicit* parameter, and the Coq code of $\mathbb{D}\texttt{ls\_}\pi_2$ witnesses structural decrease in a way suitable for a human to check at first glance.

3.3 Building an initial termination certificate

A termination certificate d for s can be computed (in the hidden realm of propositions and proofs) from the existence of x such that $\mathbb{P}\texttt{re}_{\texttt{ls}}\, s\, x$. It is easy to perform from a suitable inductive characterization of `btwn` but, in order to stick to its arithmetical definition (1), we simulate the two constructors and a special induction principle for `btwn` by proving:

$\texttt{btwn}_{\texttt{refl}}\ \{A\ n\} : \texttt{btwn}\ A\ n\ n$
$\texttt{btwn}_{\texttt{next}}\ \{A\ n\ m\} : \texttt{btwn}\ A\ n\ m \to A\ m \to \texttt{btwn}\ A\ n\ (\texttt{S}\ m)$
$\texttt{btwn}_{\texttt{ind}}\ A\ a\ b : \texttt{btwn}\ \big(\lambda n,\ A\ (\texttt{S}\ n) \to A\ n\big)\ a\ b \to A\ b \to A\ a.$

See the file `between.v` for the code of `btwn` and its tools that we use here to get

$\texttt{Lemma}\ \mathbb{P}\texttt{re}_{\texttt{ls}}\_\mathbb{D}\texttt{ls}\ \{s\} : (\exists x,\ \mathbb{P}\texttt{re}_{\texttt{ls}}\ s\ x) \to \mathbb{D}\texttt{ls}\ s$

by applying $\texttt{btwn}_{\texttt{ind}}$ to $\mathbb{D}\texttt{ls}$ and conclude with the monotonicity of `btwn`.

3.4 A tail-recursive program for the full loop

In order to prove that the output of the previous version of `loop` satisfies the desired post-condition, we could promote its type from `nat` to $\{m : \texttt{nat} \mid \mathbb{P}\texttt{ost}_{\texttt{ls}}\ s\ m\}$. This would lead to a decomposition of the result of the recursive call into a pair $\langle\!\langle m, po \rangle\!\rangle$, where m is the found value and po the associated proof of post-condition, followed by the construction of a similar pair $\langle\!\langle m, po' \rangle\!\rangle$, only different on its proof component, to be returned. A more elegant way is to proceed with proofs as with data in functional programming, when mimicking `while` loops using recursivity and accumulators. A remarkable point is that in the proofs-as-programs paradigm, proof accumulators turn out to be (proofs of) *loop invariants*, as illustrated below by b becoming $\texttt{btwn}_{\texttt{next}}\ b\ \langle t, q \rangle$ in the recursive call.

In more detail, we first fix a starting value s and take as invariant $\texttt{btwn}\ (D_{\texttt{test}} \wedge_1 Q)\ s\ n$. The linear search algorithm then calls `loop` with s as the initial input value for n, $(\mathbb{P}\texttt{re}_{\texttt{ls}}\_\mathbb{D}\texttt{ls}\ e)$ as the initial termination certificate, where e is a proof of $(\exists n,\ \mathbb{P}\texttt{re}_{\texttt{ls}}\ s\, n)$, and $\texttt{btwn}_{\texttt{refl}}$ as the initial (proof of the) invariant. In the course of the loop, we assume a proof of the invariant for n named b; the proof of $D_{\texttt{test}}\ n$ derived from d is first bound to t; in the recursive call, the (proof of the) invariant for $\texttt{S}\ n$ is derived from b, t and $q : Q\ n$ using $\texttt{btwn}_{\texttt{next}}$; and finally, when the test provides a proof $p : P\ n$, the desired proof of $\mathbb{P}\texttt{ost}_{\texttt{ls}}\ s\ n$ is just $\langle t, p \rangle$ paired with the invariant b. Altogether, the extended code of LS is rather short.

```
Fixpoint loop n (d : DDls n) (b : btwn (D_test ∧1 Q) s n) : {m | Post_ls s m} :=
  let t := DDls_π1 d in
  match test n t with
  | left p  ⇒ ⟪n, ⟨t, p, b⟩⟫
  | right q ⇒ loop (S n) (DDls_π2 d q) (btwn_next b ⟨t, q⟩)
  end.
Definition linear_search : (∃x, Pre_ls s x) → {m | Post_ls s m} :=
  λ e, loop s (Pre_ls_DDls e) btwn_refl.
```

3.5 From linear search to μ-minimization

To use the above linear search program, we only have to instantiate P, Q, $D_{\texttt{test}}$ and the `test` function, to provide a corresponding proof of `PQ_abs`, and possibly to add stubs to adapt the pre- and post-conditions. For instance in the case of Constructive Epsilon, we are

given an arbitrary predicate P on `nat`, with the hypothesis `P_dec` : $\forall n, \{P\ n\} + \{\neg P\ n\}$. Then we keep P and take $\neg P$ for Q, $(\lambda \_, \texttt{True})$ for $D_{\texttt{test}}$ and $\lambda n\, \_, \texttt{P\_dec}\ n$ for `test`; the proof of `PQ_abs` is trivial.

The case of μ-minimization is more interesting, in particular, we have a non-trivial instantiation for $D_{\texttt{test}}$. We are given a functional relation F : `nat` $\to$ `nat` $\to$ `Prop` such that, if n is in the domain of F, then the y such that $F\,n\,y$ can be computed by a program $f : \forall n, \texttt{ex}\,(F\ n) \to \texttt{sig}\,(F\ n)$. We want to transfer this to the minimization of F, that is, assuming the existence of a minimal m such that F is defined and strictly positive for all $x < m$, and $F\ m\ \texttt{O}$, we want to provide a computation returning this (unique) m. Formally, we define:

$$\texttt{def\_at}\ F := \lambda n, \exists y, F\,n\,y \qquad \texttt{ze\_at}\ F := \lambda n, F\,n\,\texttt{O} \qquad \texttt{pos\_at}\ F := \lambda n, \exists k, F\,n\,(\texttt{S}\,k)$$
$$\texttt{umin}_0\ F\ y := \texttt{ze\_at}\ F\ y \wedge \forall n, n < y \to \texttt{pos\_at}\ F\ n$$

The formal specification of μ-minimization is then $\texttt{ex}\ (\texttt{umin}_0\ F) \to \texttt{sig}\ (\texttt{umin}_0\ F)$. As a natural and cheap generalization, we also define $\texttt{umin}\ F\ s\ y := \texttt{ze\_at}\ F\ y \wedge \texttt{btwn}\ (\texttt{pos\_at}\,F)\ s\ y$. Then μ-minimization is the special case, with $s := 0$, of

$$\forall s, \texttt{ex}\ (\texttt{umin}\ F\ s) \to \texttt{sig}\ (\texttt{umin}\ F\ s) \tag{2}$$

This is very close to the specification of linear search, with $D_{\texttt{test}}$ instantiated as $\texttt{def\_at}\,F$, P as $\texttt{ze\_at}\,F$ and Q as $\texttt{pos\_at}\,F$ respectively. In order to use its implementation given in Section 3.4, we just have to feed it with a proof of `PQ_abs` (simple, using functionality of F and discrimination between `O` and `S` _) and a suitable `test` program:

```
Let test n (t : D_test n) : {P n} + {Q n} :=
  let (k, e_k) := f n t in
  match k return F _ k → _ with O ⇒ λ e, left e | S r ⇒ λ e, right ⟪r, e⟫ end e_k
```

where the term $e_k : F\,n\,k$ is analyzed as either $e : F\,n\,\texttt{O}$ (when k matches `O`) or $e : F\,n\,(\texttt{S}\,r)$ (when k matches $\texttt{S}\,r$).

However, the code obtained is this way is somewhat unsatisfactory, because each call to `loop` first constructs a dependent Boolean from the input and next immediately performs a pattern matching on the latter, and this intermediate Boolean would be reflected at extraction stage. A simple way to improve this state of affairs consists in performing a program transformation on the Coq `loop`, taking advantage, in passing, of $P \subseteq D_{\texttt{test}}$ and $Q \subseteq D_{\texttt{test}}$. Those transformations have only an impact on the loop (6 lines of code):

```
Fixpoint loop n (d : 𝔻ls D_test P n) (b : btwn Q s n) : sig (umin F s) :=
  let (k, e_k) := f n (𝔻ls_π1 D_test P d) in
  match k return F _ k → _ with
  | O   ⇒ λ e, ⟪n, ⟨e, b⟩⟫
  | S _ ⇒ λ e, loop (S n) (𝔻ls_π2 D_test … d ⟪_, e⟫) (btwn_next b ⟪_, e⟫)
  end e_k.
```

and the code of `linear_search` can be reproduced as is.

In conclusion to the file `umin_compute.v`, the program $\texttt{umin}_0\texttt{\_compute}$, which is needed in our full development of μ-recursive algorithms, is defined in two lines by functional composition of `linear_search` and simple monotonicity lemmas relating the various statements of `umin`. The reader can consult the file `umin_compute_details.v` to get a gradual derivation of the above `loop` from the one used in the original `linear_search`. As a side remark, it is

possible to present μ-minimization as just an instance of a parametric version of linear search, allowing us to share not only the $\mathbb{D}$`ls` predicate but also the code of the loop whatever the type of the result of the `test` function and the resulting extracted program. But, as symptomatic to many generic code constructions, it is unfortunately not yet as short and reasonably explainable as the above compromise between code sharing and code readability, where we transform a small part of the generic code to get μ-minimization.

4 Representing μ-algorithms in Coq

Basing on our implementation of μ-minimization, we now switch to the intentional encoding of all μ-recursive algorithms in Coq. We follow the idea already developed in [9, 10] of capturing the syntax of μ-algorithms in a type dependent on the arity (number of parameters of the corresponding function), and nested with a parametric type of dependent vectors. However, in this pearl, we insist on giving a minimized and as clean as possible account of these types.

We do not use lists at all. Instead, for a given base type X, we use *vectors* of type denoted $\texttt{vec}\,X\,n$, which are lists but augmented with a further dependency on their length, herein denoted using $n : \texttt{nat}$. Components of vectors are accessed through indices in the finite type $\texttt{idx}\,n : \texttt{Set}$ also dependent on n. In the sequel, $\langle\,\rangle$ denotes the empty vector, $a :: v$ denotes the vector made of a followed by v, and $v.[i]$ denotes the ith element of v. We also might write $\langle a; b; c\rangle$ for the vector $a :: b :: c :: \langle\,\rangle$. Our files `index.v` and `vec.v` reproduce the types `Fin.t` and `Vector.t` of the Coq standard library but are better tailored towards clean extraction. Technical details on our library are postponed to 4.4.

4.1 μ-algorithms as a nested dependent type

Usually leaving arities (denoted using the letters a, b) as implicit arguments of Coq constructors or terms, we define the type of μ-algorithm of arity a as $\texttt{recalg}\,a$ or simply $\mathcal{A}_a : \texttt{Set}$ herein, with the following constructors (inductive rules):

$$\frac{}{\texttt{ra\_zero} : \mathcal{A}_0} \qquad \frac{}{\texttt{ra\_succ} : \mathcal{A}_1} \qquad \frac{i : \texttt{idx}\,a}{\texttt{ra\_proj}\,i : \mathcal{A}_a}$$

$$\frac{f : \mathcal{A}_b \qquad g : \texttt{vec}\,\mathcal{A}_a\,b}{\texttt{ra\_comp}\,f\,g : \mathcal{A}_a} \qquad \frac{f : \mathcal{A}_a \qquad g : \mathcal{A}_{2+a}}{\texttt{ra\_prec}\,f\,g : \mathcal{A}_{1+a}} \qquad \frac{f : \mathcal{A}_{1+a}}{\texttt{ra\_umin}\,f : \mathcal{A}_a}$$

This inductive data-structure that we define in `recalg.v` mimics that in [9] but we use slightly different schemes to better match those of [1] that serve as our textbook reference here. For instance, we do not have constants (of arity 0) except for the zero constant itself which appears both at arity 0 and at arity 1 in [9].

The constructor $\texttt{ra\_comp}\,\_\,\_$ for *composition* nests the type $\mathcal{A}\_$ with the type of vectors $\texttt{vec}\,\_\,\_$, hence Coq fails to automatically derive a powerful enough eliminator for that nested inductive type. For completeness or further extensions, we provide a hand written general eliminator `recalg_rect` for $\mathcal{A}_a$ in the file `recalg.v` (as a suitable fixpoint), but we will not need it in this pearl since we will always reason or compute inductively on $\mathcal{A}_a$ using hand-written fixpoints, i.e. by inlining `recalg_rect`.

4.2 The semantics of μ-algorithms

We characterize the semantics $[\![S_a]\!]\ v_a\ o$ of the μ-algorithm of $S_a : \mathcal{A}_a$ by interpreting it as a binary relation between an input vector $v_a : \texttt{vec}\,\texttt{nat}\,a$ and an output value in $o : \texttt{nat}$, hence for $\mathcal{IO}_a := \texttt{vec}\,\texttt{nat}\,a \to \texttt{nat} \to \texttt{Prop}$, we seek to define $[\![S_a]\!] : \mathcal{IO}_a$. The denotation S_a is intended

to recall that this is *S(ource code) for arity a*. Notice that this provides an extensional meaning to S_a that restricts the possible algorithmic interpretations (or intentional meaning) of S_a which must realize that specification. To do so, we define $\texttt{ra\_sem} : \forall\{a\}, \mathcal{A}_a \to \mathcal{IO}_a$ as a fixpoint where we denote $[\![S_a]\!] := \texttt{ra\_sem}\, S_a$:

```
Fixpoint ra_sem {a} (S_a : A_a) : IO_a :=
  match S_a with
  | ra_zero   ⇒ Zr       | ra_comp S_b S_ab  ⇒ Cn [[S_b]] (vec_map [[·]] S_ab)
  | ra_succ   ⇒ Sc       | ra_prec S_a S_a'' ⇒ Pr [[S_a]] [[S_a'']]
  | ra_proj i ⇒ Id i      | ra_umin S_a'      ⇒ Mn [[S_a']]
  end where [[S_a]] := (ra_sem S_a).
```

Notice the nesting of `vec_map` which applies `ra_sem`, to every component of the vector $S_{ab} : \texttt{vec}\, \mathcal{A}_a\, b$, and the references to `Zr`, `Sc`, `Id`, `Cn`, `Pr`, `Mn` which correspond to the Coq encoding of μ-recursive schemes as defined in [1]. The fixpoint proceeds by structural induction on S_a but the guard-checker *inspects the code* of the nested instance of `vec_map` to ensure $[\![\cdot]\!]$ is called only on sub-terms of the vector S_{ab}. Notice that since the inductive type $\mathcal{A}_a$ nests the type of vectors $\texttt{vec}\, \mathcal{A}_a\, b$ in the constructor $\texttt{ra\_comp}\, \_\, \_$, the only way to traverse such structures is via nested fixpoints which, if properly written, can fortunately be accepted by the guard condition of the type-checker.

As a side note, instead of a direct fixpoint, we could use the general recursor `recalg_rect` (see file `recalg.v`), but this would have unfortunate consequences: it would hinder a unified presentation of `ra_sem` and `ra_compute`. Indeed, the induction hypothesis for the `ra_comp` constructor does not expect a vector but a (dependent) map, and would thus be incompatible with the output type of `vec_map`, hence involving some glue code. That glue code would also be necessary in the upcoming fixpoint `ra_compute` in section 4.3, and would there unfortunately reflect in the extracted OCaml code. To get a unified presentation of `ra_sem` and `ra_compute`, we choose to inline `recalg_rect` in both cases.

We follow precisely our reference textbook [1, p. 63] using reversely ordered vectors to represent tuples. See file `recalg_semantics.v` where we define

```
Definition Zr : IO_0 := λ _ y, y = 0.
Definition Sc : IO_1 := λ v_1 y, y = 1 + vec_head v_1.
Definition Id {a} (i : idx a) : IO_a := λ v_a y, y = v_a.[i].
```

We follow up with composition of a b-ary μ-algorithm with a vector of a-ary μ-algorithms:

```
Definition Cn {a b} (φ_b : IO_b) (ψ_ab : vec IO_a b) : IO_a :=
          λ v_a y, ∃v_b, φ_b v_b y ∧ ∀i, ψ_ab.[i] v_a v_b.[i]
```

to be found in [1, p. 64]. Primitive recursion is mechanically best described using a higher-order primitive recursive scheme (like that of e.g. Gödel system T). We define

```
Definition Pr {a} (φ_a : IO_a) (ψ_a'' : IO_2+a) : IO_1+a :=
          vec_S_inv (λ n v_a, prim_rec φ_a (λ w_a m y, ψ_a'' (m :: y :: w_a)) v_a n)
```

where `prim_rec` is defined in `schemes.v` as the following instance `nat_rect`, the dependent eliminator/recursor for the `nat` type and $\diamond$ denotes the right-associative composition of a binary relation with a unary one:

```
Context {X Y : Type} (F : X → Y → Prop) (G : X → nat → Y → Y → Prop).
Definition prim_rec (x_0 : X) := nat_rect (λ _, Y → Prop) (F x_0) (λ n p, (G x_0 n) ◇ p).
```

Informally, this would read as `prim_rec` $x_0\ n := \big(G\,x_0\,(n-1)\big) \diamond \cdots \diamond (G\,x_0\,0) \diamond (F\,x_0)$. We check that `Pr` satisfies the following two definitional equations, which correspond to the characterization of primitive recursive scheme in [1, p. 67].

1. `Pr` $\varphi_a\ \psi_{a''}\ (\mathtt{O} :: v_a)\ y = \varphi_a\ v_a\ y$;
2. `Pr` $\varphi_a\ \psi_{a''}\ (\mathtt{S}\,n :: v_a)\ y = \exists o,$ `Pr` $\varphi_a\ \psi_{a''}\ (n :: v_a)\ o \wedge \psi_{a''}\ (n :: o :: v_a)\ y$.

We finish with the scheme of unbounded minimization (starting at `O`), defined via a more general scheme of minimization starting at a value given as extra parameter:

```
Definition Mn {a} (φ_{a'} : 𝓘𝓞_{1+a}) : 𝓘𝓞_a := λ v_a, umin_0 (λ y, φ_{a'}(y :: v_a)).
```

The definitions of `umin` and $\mathtt{umin}_0$ occur in the file `schemes.v` and are also discussed in Section 3.5. We check that `Mn` satisfies the following definitional equation:

$$\mathtt{Mn}\ \varphi_{a'}\ v_a\ y\ =\ \varphi_{a'}\ (y :: v_a)\ \mathtt{O} \wedge \forall n, n < y \to \exists k, \varphi_{a'}\ (n :: v_a)\ (\mathtt{S}\,k)$$

which mimics the definition of the minimization scheme in [1, p. 70].

Having defined the semantic (extensional) interpretation of μ-algorithms, we verify that $[\![S_a]\!]$ is a functional relation. We proceed by structural induction on S_a,

```
Theorem ra_sem_fun {a} (S_a : 𝓐_a) : functional [[S_a]]
```

directly with a fixpoint, by compositionally exploiting the fact that μ-recursive schemes preserve functional relations.

4.3 The interpreter for μ-algorithms

Following the approach hinted in the introduction, given a specification predicate $P : X \to$ `Prop` over a type X, we characterize a *specified partial value* by a term $t : (\exists x, P\,x) \to \{x \mid P\,x\}$ in which the specified value $\{x \mid P\,x\}$, that is, an x paired with a proof of $P\,x$, is guarded by its existence $(\exists x, P\,x)$:

- The unary predicate $P : X \to$ `Prop` gives the specification of the value. In our case, we instantiate e.g. $P := [\![S_a]\!]\ v_a :$ `nat` $\to$ `Prop`, hence, thanks to `ra_sem_fun`, P will hold for at most one x;
- the Coq term t computes a value x such that $P\,x$, provided it is given a certificate for its algorithm to terminate the computation, stated as the non-informative existence of an (output) value satisfying P.

In the file `compute_def.v`, we define the predicate capturing specified partial values as `compute` $\{X\}\ (P : X \to$ `Prop`$) := (\exists x, P\,x) \to \{x \mid P\,x\}$.[2] This encoding of partiality allows a direct generalization of the code of the semantic `ra_sem` (noted $[\![\cdot]\!]$) into an interpreter `ra_compute` (noted $[\![\cdot]\!]_o$) as described below, with relatively short proof terms for pre/post conditions.

As a side note, our approach contrasts with the "partiality monad" of [2] where a partial value is of type $\{Q :$ `Prop` $\mid Q \to X\}$, i.e. though guarded by a predicate Q, it refers to an output type X *only*, and is not further specified in that type. In our case, a `compute` value is always specified w.r.t. a predicate over a type (i.e. P) which in practice characterizes that value uniquely. It comes with a proof that the value satisfies its specification (i.e. $P\,x$). Hence that guard and the specification are linked together. Also, having an output specification allows us to compositionally derive a correct-by-construction interpreter.

[2] Notice that this definition is not intended to hint at the traditional notion of computability.

We can now present the Coq term for μ-algorithms that is going to be extracted into OCaml as a natural interpreter for μ-algorithms in that programming language. For $S_a : \mathcal{A}_a$, the term `ra_compute` S_a : $\forall v_a$: `vec nat` a, `compute` $\big([\![S_a]\!]\, v_a\big)$ will realize the extensional interpretation $[\![\cdot]\!]$, by directly computing the output from the input along the lines of the given μ-algorithm. In the file `interpreter.v`, we write the following `Fixpoint ra_compute` $\{a\}$ S_a also denoted $[\![S_a]\!]_{\mathrm{o}}$ (for a more compact notation) reusing the same scheme as that of the code of `ra_sem` $\{a\}$ $S_a = [\![S_a]\!]$. The suffix in the new notation $[\![\cdot]\!]_{\mathrm{o}}$ is intended to recall that we do not simply define a proposition but instead, an o(utput) value is now computed:

```
Fixpoint ra_compute {a} (S_a : A_a) {struct S_a} : ∀v_a : vec nat a, compute ([[S_a]] v_a) :=
  match S_a with
  | ra_zero       ⇒ Zr_compute
  | ra_succ       ⇒ Sc_compute
  | ra_proj i     ⇒ Id_compute i
  | ra_comp S_b S_ab ⇒ Cn_compute [[S_b]]_o (λ v_a dv_a, vec_map_compute ([·]_o v_a) S_ab dv_a)
  | ra_prec S_a S_a'' ⇒ Pr_compute [[S_a]]_o [[S_a'']]_o
  | ra_umin S_a'  ⇒ Mn_compute [[S_a']]_o
  end where [[S_a]]_o := (ra_compute S_a).
```

The sub-term $[\![\cdot]\!]_{\mathrm{o}}\, v_a$ has type $\forall S_a$, `compute` $\big([\![S_a]\!]\, v_a\big)$ which states that $[\![\cdot]\!]\, v_a$ is a partial function, which can be computed in Coq if fed with a certificate that its output exists (termination), and it is passed to `vec_map_compute` which generalize `vec_map` to the application of partial functions on every component of a vector, but still by *structural recursion* on the vector. Then the computation follows a natural interpretation of μ-algorithms as functional programs via extraction.

We further comment on the code of the `ra_compute` fixpoint, focusing on how Coq establishes termination. First, it proceeds by structural recursion on S_a. Hence the guard-checker verifies that `ra_compute` is only applied to sub-terms of S_a. And for this, it has to inspect the code of `vec_map_compute` which inevitably nests a call to `ra_compute` (noted $[\![\cdot]\!]_{\mathrm{o}}$) because S_{ab} : `vec` $\mathcal{A}_a\, b$ is a (nested) vector of sub μ-algorithms in $\mathcal{A}_a$. Because `vec_map_compute` proceeds by recursion on S_{ab}, the guard checker accepts this nesting.

In the case of `ra_umin` $S_{a'}$, we see that `Mn_compute` receives $[\![S_{a'}]\!]_{\mathrm{o}}$, the fixpoint itself applied to $S_{a'}$, as first parameter, which obviously passes the guard-checker. The code of `Mn_compute` can be found in the `compute.v` and is based on that of `umin`$_0$`_compute`; see the file `umin_compute.v` and explanations in Section 3.5.

```
Variables (a : nat) (S_a' : A_{1+a}) (cS_a' : ∀v_a', compute ([[S_a']] v_a')).
Definition Mn_compute v_a : compute (Mn [[S_a']] v_a) :=
          umin_0_compute (λ _, ra_sem_fun _ _) (λ n dn, cS_a' (n :: v_a) dn).
```

The case of `ra_prec` $S_a\, S_{a''}$ is similar to that of `ra_umin` $S_{a'}$. The case of `ra_comp` $S_b\, S_{ab}$ is however more complicated because of the nesting with `vec_map_compute` that is mandated to recursively iterate $[\![\cdot]\!]_{\mathrm{o}}\, v_a$ over the components of the vector of sub μ-algorithms S_{ab}.

```
Variables (a b : nat) (S_b : A_b) (cS_b : ∀v_b, compute ([[S_b]] v_b))
          (S_ab : vec A_a b) (cS_ab : ∀v_a, compute (λ v_b, ∀i, [[S_ab.[i]]] v_a v_b.[i])).
Definition Cn_compute : ∀v_a, compute (Cn [[S_b]] (vec_map [·] S_ab) v_a) :=
          λ v_a dv_a, let (v_b, v_a v_b) := cS_ab v_a _ in let (y, v_b y) := cS_b v_b _ in ⟨y, _⟩.
```

We leave out as holes _ the three (small) proof obligations that can be studied further in code file `compute.v`. The term `vec_map_compute` is used to fill the argument cS_{ab} of `Cn_compute` and its code is described in the file `map_compute.v`. It generalizes the code of `vec_map` to deal with specifications (i.e. pre/post conditions), but extracts the same.

4.4 Remarks on a carefully crafted library of indices and vectors

The types for indices and vectors are defined inductively with the following rules/constructors:

$$\texttt{idx} : \texttt{nat} \to \texttt{Type} := \mathbb{O} : \forall\{n\}, \texttt{idx}\,(\texttt{S}\,n) \mid \mathbb{S} : \forall\{n\}, \texttt{idx}\,n \to \texttt{idx}\,(\texttt{S}\,n)$$
$$\texttt{vec}\,X : \texttt{nat} \to \texttt{Type} := \langle\,\rangle : \texttt{vec}\,X\,\texttt{O} \mid \_ :: \_ : \forall\{n\}, X \to \texttt{vec}\,X\,n \to \texttt{vec}\,X\,(\texttt{S}\,n)$$

We can analyze the content of vectors by standard pattern matching, or, on nonempty vectors, using the standard `vec_head` : $\texttt{vec}\,X\,(\texttt{S}\,n) \to X$ and `vec_tail` : $\texttt{vec}\,X\,(\texttt{S}\,n) \to \texttt{vec}\,X\,n$ functions. But to access the components in a more versatile way, as sometimes required by the definition of μ-algorithms, we define the projection $\texttt{vec\_prj}\,\{X\,n\} : \texttt{vec}\,X\,n \to \texttt{idx}\,n \to X$. The Coq fixpoint defining `vec_prj` is carefully written by structural recursion on the vector,[3] and the use of $\texttt{idx\_inv}\,\{0\} : \texttt{idx}\,0 \to \texttt{False}$ allows to dispose of the impossible case in a guard-checker friendly way.

```
Fixpoint vec_prj {n} (u : vec X n) : idx n → X :=
  match u in vec _ m return idx m → X with
  | ⟨⟩     ⇒ λ i, match idx_inv i with end
  | x :: v ⇒ λ i, match i in idx m return vec _ (pred m) → X with
                  | 𝕆   ⇒ λ _, x
                  | 𝕊 j ⇒ λ v, vec_prj v j
                  end v
  end
```

The type of $\texttt{idx\_inv}\,\{n : \texttt{nat}\}$ is a bit more general (by dependent pattern matching on n), to also allow inversions of indices in $\texttt{idx}\,(\texttt{S}\,n)$ but the idea is the same. Actually, besides the definition of $\texttt{idx}\,n$, the statement of the lemma `idx_inv` together with its short proof is the only tool defined in our library for indices (in file `index.v`). Notice that we avoid `idx_inv` by inlining it in the second match case $x :: v$ because using it would introduce an additional level of constructors/matches in the extracted code.

Then $v.[i]$ is just a convenient notation for $\texttt{vec\_prj}\,v\,i$. As a consequence of our definition, the identities $(x :: v).[\mathbb{O}] = x$ and $(x :: v).[\mathbb{S}\,i] = v.[i]$ hold by definitional equality. But more importantly, any component $v.[i]$ is recognized as a *sub-term* of v by the guard-checker when type-checking a fixpoint nesting a call to `vec_prj`. Additionally, `vec_prj` extracts to desirable OCaml code:

```
let rec vec_prj u i = match u with
  | ⟨⟩     → assert false
  | x :: v → match i with O → x | S j → vec_prj v j
```

The projection `vec_prj` allows to view the inductive type $\texttt{vec}\,X\,n$ as an extensional representation of the type $\texttt{idx}\,n \to X$: two vectors are equal iff their components are equal, i.e. $\big(\forall i,\, v.[i] = w.[i]\big) \to v = w$, which is not the case for "functional vectors" in $\texttt{idx}\,n \to X$.

Complementary to `vec_head` and `vec_tail`, we also provide versatile inversion lemmas for vectors in either $\texttt{vec}\,X\,\texttt{O}$ or $\texttt{vec}\,X\,(\texttt{S}\,n)$ of types (see `vec.v` for detailed explanations):

$$\texttt{Definition vec\_O\_inv}\ \{X\}\ \{P : \texttt{vec}\,X\,\texttt{O} \to \texttt{Type}\} : P\,\langle\,\rangle \to \forall u,\, P\,u.$$
$$\texttt{Definition vec\_S\_inv}\ \{X\,n\}\ \{P : \texttt{vec}\,X\,(\texttt{S}\,n) \to \texttt{Type}\} : \big(\forall x\,v, P\,(x :: v)\big) \to \forall u, P\,u.$$

[3] The Coq standard library version `8.16.1` of `Vector.nth` proceeds by recursion on the index, not on the vector, which would conflict with our guard conditions, but a version of `nth` computationally similar to `vec_prj` has been accepted into future revisions of the standard library as PR #16731.

For instance, the term `vec_S_inv` $(\lambda\, x\, v,\ f\, x\, v)\, u$ can then be seen as a correct (hence type-checkable) way to write something like `match` u `with` $x :: v \Rightarrow f\, x\, v$ `end` for a vector u : `vec` _ (`S` _), that moreover *extracts* into a pattern-matching on u, i.e. of the form

`match` u `with` $\langle\,\rangle \to$ `assert false` $\mid x :: v \to f\, x\, v$.

The alternate code f (`vec_head` u) (`vec_tail` u) would extract less gracefully in two successive pattern-matchings on u performed inside `vec_head` and `vec_tail`.

5 Extraction to OCaml

To shorten a bit the extracted code and make it easier to read, in the file `interpreter.v` we feed the extraction plugin of Coq with several kinds of directives:

- we generally forget about arities because they do not participate in the computation. They exist at proof-level to ensure that e.g. composition (resp. projection) occurs only between vectors (resp. and indices) of proper arities;
- we extract indices in `idx` n as natural numbers directly to avoid duplication of code between `idx` _ and `nat`;
- having forgotten their arity, we can extract vectors as native OCaml lists to present the reader with a familiar notation for tuples;
- we inline some Coq terms to avoid duplicating OCaml names for the same functions and avoid steps that need no factorization because they are only used once.

With those directives, in a first iteration of extraction, we get the OCaml interpreter for μ-algorithms as presented in the introduction, with two minor differences that we describe and discuss how they can be addressed below.

The first difference is that Coq does not generate partial `match` filters and thus, we get extra `assert false` statements instead of missing match cases. Computationally the only difference this makes is in the name of the generated exception. However, they should not be triggered unless the OCaml interpreter is called on a context which could not be typed within Coq, e.g. the input vector has a shorter length that the arity of the μ-algorithm.[4]

The second difference is the occurrence of `__` = `Obj.t` OCaml type and object that the extraction plugin uses to circumvent the type system of OCaml on Coq types which are too general for it.[5] This difference is more important to tackle in our opinion.

Let us start with an explanation of why these `__` appear in the extracted code in the first place. They come from e.g. the second (non implicit) argument of $\texttt{umin}_0\texttt{\_compute}$, which is a partial value of type $f : \forall n, \texttt{compute}\ (F\, n)$ or, by expanding the definition of `compute`, of type $f : \forall n, (\exists y, F\, n\, y) \to \{y \mid F\, n\, y\}$. The extraction plugin is not able to recognize that it can safely erase $(\exists y, F\, n\, y)$ because f is itself *an argument* of $\texttt{umin}_0\texttt{\_compute}$. No directive of our knowledge is able to inform the extraction plugin with non-informative data in the types of the arguments of extracted terms.

We present two ways of getting rid of `__`. Both consist in hiding the proposition $(\exists y, F\, n\, y)$ in the propositional part of a Σ-type. We think it is better to describe these tricks as a diff on the Coq code rather than directly exposing a more convoluted variant of the interpreter; see files `unit.diff` and `hide.diff`.

[4] We do not know if it is possible to instruct the extraction plugin to dismiss impossible match cases.

[5] Additional issues could be raised by the call-by-value evaluation strategy of OCaml; anyway, extracting to Haskell produces similar extra arguments of type `any`, showing that the discussion below still makes sense even with a lazy strategy, especially the second improvement.

The first approach consists in adding a new parameter of type `unit` and packing it with the proposition $(\exists y, F\,n\,y)$, hence we get the following definition of $\texttt{compute}_u$:

$$\texttt{Definition compute}_u\ \{X\}\ (P : X \to \texttt{Prop}) := \{\_ : \texttt{unit} \mid \exists x, P\,x\} \to \{x \mid P\,x\}$$

the type of the parameter f in $\texttt{umin}_0\texttt{\_compute}$ becoming $f : \forall n, \texttt{compute}_u\,(F\,n)$. We do not need to upgrade `compute` into $\texttt{compute}_u$ everywhere though, only when a parameter is a partial function, e.g. in the definitions of `prim_rec_compute`, or `vec_map_compute` or that of `umin_compute` and $\texttt{umin}_0\texttt{\_compute}$.

This solution has the advantage of symmetry (see below) and conceptual simplicity. The simplest way to visualize small amount of needed updates in the code is through the diff file `unit.diff`. The resulting extracted code is the same as in the first iteration except that `Obj.t` (resp. `__`) gets substituted with `unit` (resp. `()`). So there is no trick to circumvent the OCaml type system anymore but still, an extra dummy/`unit` argument remains.

The second approach gives us an extraction where the `__` parts are completely removed from the code. This is quite satisfying and not much more complicated than the `unit` trick but we lose symmetry in the treatment of the arguments of Coq terms. The trick consists in hiding $(\exists y, F\,n\,y)$ directly under the *last argument* it depends on, hence n in the case of $\texttt{umin}_0\texttt{\_compute}$. So we get the following type for its second argument:

$$f : \forall p : \{n \mid \exists y, F\,n\,y\},\ \{y \mid F\ (\pi_1\,p)\ y\}.$$

Again, we only need to make that change on the type of the arguments that represent partial functions, e.g. f, not on the terms implementing partial functions, e.g. $\texttt{umin}_0\texttt{\_compute}$. We recall that the simplest way to visualize the small amount of required modifications in the Coq code is via the diff file `hide.diff`.

6 Conclusion

Program extraction was advocated as an interesting approach to the study of the correctness (by construction) of functional programs for a long time, and the issue of partial functions, especially possibly non-terminating programs, was raised very early, both in untyped settings such as PX [6] and in strongly typed logical settings where only terminating (functional) programs can be expressed such as Nuprl [3]. Parametric ways to deal with partial values in Coq include [2, 4], allowing for the development of synthetic computability theory [5].

In addition to theoretical considerations, the issue of partiality is not that easy from a practical point of view, notably when partial functions are mixed with higher-order functional programs: when the latter are basically structurally recursive, it is desirable to keep their conceptual simplicity as much as possible.

We think that the example of μ-recursive functions contains a significant summary of the issues raised, so the work presented here may help to understand how they can be dealt with in CIC as implemented in the Coq proof assistant. We could have just tried to follow the Braga method [11], i.e., to provide an inductive definition of the domain of the *full* desired interpreter (`ra_compute`) either directly, or from an inductive presentation of its input-output graph. Clearly, the resulting development would have been much more convoluted. Instead, we have limited the use of the machinery of [11] – actually a tail-recursive variant of it – at the single place where it is relevant (unbounded minimization), using ordinary structural recursion on the input data at all other places. Beyond careful explanations on why and how guard condition for termination are satisfied, the resulting development is made conceptually

simple and concise. To this effect, a very simple, hence easy to overlook idea turned out to be surprisingly effective: use $\forall x,\ (\exists y, G\,x\,y) \to \{y \mid G\,x\,y\}$ as a general shape for specifying `ra_compute` and its auxiliary functions.

Note that, as a bonus, the results previously presented in [9] can then be obtained with much shorter and elegant proofs.

References

1. George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, 4 edition, 2002. doi:10.1017/CBO9781139164931.
2. Mario Carneiro. Formalizing Computability Theory via Partial Recursive Functions. In *ITP 2019*, volume 141, pages 12:1–12:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.12.
3. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
4. Yannick Forster. Church's Thesis and Related Axioms in Coq's Type Theory. In *CSL 2021*, volume 183, pages 21:1–21:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CSL.2021.21.
5. Yannick Forster. Parametric church's thesis: Synthetic computability without choice. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, pages 70–89, Cham, 2022. Springer International Publishing.
6. Susumu Hayashi. *Extracting Lisp Programs from Constructive Proofs: A Formal Theory of Constructive Mathematic Based on Lisp*, volume 19, pages 169–191. Publications of the Research Institute for Mathematical Sciences, 1983.
7. Stephen C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340–353, 1936. doi:10.1215/S0012-7094-36-00227-2.
8. Stephen C. Kleene. Recursive predicates and quantifiers. *Trans. Amer. Math. Soc.*, 53:43–73, 1943. doi:10.1090/S0002-9947-1943-0007371-8.
9. Dominique Larchey-Wendling. Typing Total Recursive Functions in Coq. In *ITP 2017*, pages 371–388. Springer, 2017. doi:10.1007/978-3-319-66107-0_24.
10. Dominique Larchey-Wendling and Yannick Forster. Hilbert's Tenth Problem in Coq (Extended Version). *Logical Methods in Computer Science*, Volume 18, Issue 1:35:1–35:41, March 2022. doi:10.46298/lmcs-18(1:35)2022.
11. Dominique Larchey-Wendling and Jean-François Monin. *The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq*, chapter 8, pages 305–386. World Scientific, 2021. doi:10.1142/9789811236488_0008.
12. Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley series in logic. Addison-Wesley, 1967.
13. Vincent Zammit. A Proof of the S-m-n theorem in Coq. Technical report, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK, March 1997. URL: http://kar.kent.ac.uk/21524/.