# Typing Total Recursive Functions in Coq

Dominique Larchey-Wendling(✉)

LORIA – CNRS, Nancy, France
dominique.larchey-wendling@loria.fr

**Abstract.** We present a (relatively) short mechanized proof that Coq types any recursive function which is provably total in Coq. The well-founded (and terminating) induction scheme, which is the foundation of Coq recursion, is maximal. We implement an unbounded minimization scheme for decidable predicates. It can also be used to reify a whole category of undecidable predicates. This development is purely constructive and requires no axiom. Hence it can be integrated into any project that might assume additional axioms.

## 1 Introduction

This paper contains a mechanization in Coq of the result that any total recursive function can be represented by a Coq term. A short slogan could be *Coq types any total recursive function,* but that would be a bit misleading because the term *total* might also refer to the *meta-theoretical level* (see Sect. 7).

The theory of partial recursive (or $\mu$-recursive) functions describes the class of recursive functions by an inductive scheme: it is the least set of partial functions $\mathbb{N}^k \longrightarrow \mathbb{N}$ containing constant functions, zero, successor and closed under composition, recursion and *unbounded minimization* [9]. Forbidding minimization (implemented by the $\mu$ operator) leads to the sub-class of primitive recursive functions which are total functions $\mathbb{N}^k \longrightarrow \mathbb{N}$. Coq has all the recursive schemes except unbounded minimization so it is relatively straightforward to show that any primitive recursive function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ can be represented by a Coq term $t_f : \mathcal{N}^k \to \mathcal{N}$ where $\mathcal{N}$ is a short notation for the Coq type `nat` of Peano natural numbers. To represent all partial recursive functions $\mathbb{N}^k \longrightarrow \mathbb{N}$ by Coq terms, we would first need to deal with partiality and change the type into $\mathcal{N}^k \to \texttt{option}\ \mathcal{N}$ (for instance) because (axiom-free) Coq only contains total functions; so here the term $\texttt{None} : \texttt{option}\ \mathcal{N}$ represents the undefined value. Unfortunately, this does not work because Coq (axiom-free) meta-level normalization would transform such an encoding into a solution of the *Halting problem*.

Then, from a theoretical standpoint one question remains: which are the functions that Coq can represent in the type $\mathcal{N}^k \to \mathcal{N}$. In this paper, we give a mechanized proof that formally answers of half of the question:

> *The type $\mathcal{N}^k \to \mathcal{N}$ contains every recursive function of arity $k$ which can be proved total in Coq.*

Work partially supported by the TICAMORE project (ANR grant 16-CE91-0002).


M. Ayala-Rincón and C.A. Muñoz (Eds.): ITP 2017, LNCS 10499, pp. 371–388, 2017.
DOI: 10.1007/978-3-319-66107-0_24

Such a result was hinted in [2] but we believe that mechanizing the suggested approach implies a lot of work (see Sect. 2). This property of totality of Coq can compared to the characterization of System F definable functions as those which are provably total in $\mathrm{AF}_2$ [5]. Besides the fact that $\mathrm{AF}_2$ and Coq are different logical frameworks, the main difference here is that we mechanize the result inside of Coq itself whereas the $\mathrm{AF}_2$ characterization is proved at the meta-theoretical level.

Before the detailed description of our contributions, we want to insist on different meanings of the *notion of function* that should not be confused:

- The $\mu$-recursive schemes are the constructors of an inductive type of *algorithms* which are the "source code" and can be interpreted as partial function $\mathbb{N}^k \longrightarrow \mathbb{N}$ in Set theory or as predicates $\mathcal{N}^k \to \mathcal{N} \to \texttt{Prop}$ in Coq;
- The *Set-theoretic notion of partial function* is a *graph/relation* between elements and their images. $\mu$-recursive functions should not be understood independently of the algorithm that implements theses relations: it is impossible to recover an algorithm from the data of the graph alone;
- Then Coq has *function types* $A \to B$ which is a related but nevertheless entirely different notion of function and we rather call them predicates here.

Now let us give a more detailed description of the result we have obtained. We define a dependent family of types $\mathcal{A}_k$ representing recursive algorithms of arity $k : \mathcal{N}$. An algorithm $f : \mathcal{A}_k$ defines a (partial) recursive function denoted $[\![f]\!]$ and which is represented in Coq as a predicate $[\![f]\!] : \mathcal{N}^k \to \mathcal{N} \to \texttt{Prop}$:

> *The proposition* $[\![f]\!]\ \boldsymbol{v}\ x$ *reads as: the computation of the algorithm* $f$ *from the input* $k$*-tuple* $\boldsymbol{v}$ *terminates and results in* $x$*.*

The implementation of the relation $[\![f]\!]$ is a simple exercise. It is more difficult to show that whenever the relation $(\boldsymbol{v}, x) \mapsto [\![f]\!]\ \boldsymbol{v}\ x$ between the input $\boldsymbol{v}$ and the result $x$ is *total*, then there is a term $t_f : \mathcal{N}^k \to \mathcal{N}$ (effectively computable from $f$) such that the result of the computation of $f$ on the input $\boldsymbol{v}$ is $(t_f\ \boldsymbol{v})$ for any $\boldsymbol{v} : \mathcal{N}^k$. This is precisely what we show in the following formal statement:

$$\forall (k : \mathcal{N})(f : \mathcal{A}_k),\ \left(\forall \boldsymbol{v}, \exists x, [\![f]\!]\ \boldsymbol{v}\ x\right) \to \left\{t_f : \mathcal{N}^k \to \mathcal{N} \mid \forall \boldsymbol{v}, [\![f]\!]\ \boldsymbol{v}\ (t_f\ \boldsymbol{v})\right\} \quad \text{(CiT)}$$

The statement means that if $[\![f]\!]$ represents a *total function* $(\forall \boldsymbol{v}, \exists x, [\![f]\!]\ \boldsymbol{v}\ x)$, then it can be *effectively transformed* into a Coq term $t_f : \mathcal{N}^k \to \mathcal{N}$ such that $(t_f\ \boldsymbol{v})$ is the value computed by the recursive function $[\![f]\!]$ on the input $\boldsymbol{v}$.

As we already pointed out, "Coq is Total" (CiT) is only one half of the characterization of the predicates that are definable in the type $\mathcal{N}^k \to \mathcal{N}$. The other half of the characterization, i.e. any predicate of type $\mathcal{N}^k \to \mathcal{N}$ corresponds to a $\mu$-recursive function, while meta-theoretically provable for axiom-free Coq, cannot not be proved within Coq itself; see Sect. 7.

We will call *reification* the process of transforming a non-informative predicate like $P : \forall \boldsymbol{v}, \exists x, [\![f]\!]\ \boldsymbol{v}\ x$ into an informative predicate $Q : \forall \boldsymbol{v}, \{x \mid [\![f]\!]\ \boldsymbol{v}\ x\}$.[1]

[1] From which the term $t_f := \boldsymbol{v} \mapsto \texttt{proj1\_sig}(Q\ \boldsymbol{v})$ of (CiT) is trivially derived.

In its general form, reification is a map from $\texttt{inhabited}\, X : \texttt{Prop}$ to $X : \texttt{Type}$; it transforms a non-informative proof of existence of a witness into an effective witness. In a proof system like HOL for instance, reification is built-in by Hilbert's epsilon operator. On the contrary, because of its constructive design, Coq does not allow unrestricted reification. If needed in its full generality, it requires the addition of specific axioms as discussed in Sects. 3.1 and 8.

One of the originalities of this work is that the proof we develop is purely constructive (axiom free) and *avoids the detour through small-step operational semantics*, that is the use of a model of computation on an encoded representation of recursive functions. For instance, programs are represented by numbers (Gödel coding) in the proof of the $S$-$m$-$n$ theorem [13]. It is also possible to use other models of computations such as register machines (or Turing machines) or even $\lambda$-calculus as in [8] or in our own dependently typed implementation [7] of Krivine's reference textbook [6]; see Sect. 7.

In Sect. 2, we present an overview of the consequences of the use of small-step operational semantics and how we avoid it. In Sect. 3 we describe how to implement unbounded minimization of inhabited decidable predicates in Coq. Section 4 presents the inductive types we need for our development, most notably the dependent type $\mathcal{A}_k$ of recursive algorithms of arity $k$ and Sect. 5 defines three different but equivalent semantics for $\mathcal{A}_k$, in particular a decidable *cost aware big-step semantics* which is the critical ingredient to avoid small-step semantics. Section 6 concludes with the formal statement of (CiT) and its proof. In Sect. 7, we discuss related work and/or alternative approaches. In Sect. 8, we describe how to reify undecidable predicates (under some assumptions of course), in particular, *provability* predicates, *normalizability* predicates and even arbitrary *recursively enumerable predicates*. Section 9 lists some details of the implementation and how it is split into different libraries.

To shorten notations, we recall that we denote by $\mathcal{N}$ the inductively defined Coq type `nat` of natural numbers. The $\mu$-recursive scheme of composition requires the use of $k$-tuples which we implement as vectors. Vectors are typeset in a bold font such as in $\boldsymbol{v} : \mathcal{N}^k$ and they correspond to a polymorphic dependent type described in Sect. 4. $\Pi$-types are denoted with a $\forall$ symbol. We denote $\Sigma$-types with their usual Coq notations, which are $(\exists x, P\ x) : \texttt{Prop}$ for non-informative existential quantification, $\{x \mid P\ x\} : \texttt{Set}$ for informative existential quantification, or even $\{x : X\ \&\ P\ x\} : \texttt{Type}$ when $P : X \to \texttt{Type}$ carries information as well. These $\Sigma$-types are inductively defined in modules `Logic` and `Specif` of the standard library. The interpretation of the different existential quantifiers of Coq is discussed in Sect. 3.1.

## 2 Avoiding Small-Step Operational Semantics

In this section we give a high level view of our strategy to obtain a mechanized proof of the typability of total recursive functions in Coq. Let us first discuss the approach which is outlined in [2] (Sect. 4.4, p. 685).

1. By *Kleene's normal form theorem* [9], every recursive function can be obtained by the minimization of a primitive (hence total) recursive function;
2. Every primitive recursive function can directly be typed in Coq. The primitive recursion scheme is precisely the recursor `nat_rec` corresponding to the inductive type `nat` (denoted $\mathcal{N}$ in this paper);
3. The outermost minimization could be implemented by a "specific minimization function" defined by mutual structural recursion.

Items 1 and 2 are results which should not come as a surprise to anyone knowledgeable of $\mu$-recursion theory and basic Coq programming. These observations were already made in [2]. Their approach to minimization (i.e. Item 3) seems[2] however distinct from what we propose as Item 3′ here:

3′. Minimizations of inhabited and decidable predicates of type $\mathcal{N} \to$ `Prop` can be implemented in (axiom free) Coq.

Item 3′ could be considered as a bit surprising. Indeed, inductive type theory and hence Coq prohibits unbounded minimization. Hence we did not suspect that Coq could have such a property. When it first came to our attention, we realized that it provided a direct path towards a proof that Coq "had" any total recursive function. Critical for our approach, Item 3′ is described in Sect. 3.

Despite its apparent straightforwardness, this three steps approach (with either Item 3 or Item 3′) is difficult to implement because of Item 1. Indeed, let us describe more precisely what it implies. Kleene's normal form theorem involves the $T$ primitive recursive predicate which decides whether a given (encoding of a) computation corresponds to a given (encoding of a) program code or not. For this, you need a *small-step operational semantics* (a model of computation), say for instance Minsky (or counter) machines, and a compiler from recursive functions code to Minsky machines. You need of course a correctness proof for that compiler. Since the $T$ predicate operates on natural numbers $\mathcal{N}$, all these data-structures should be encoded in $\mathcal{N}$ which complicates proofs further. Then the $T$ predicate should answer the following question: does this given encoding of a sequence of states correspond to the execution of that given encoding of a Minsky machine. Most importantly, the $T$ predicate should be proved primitive recursive and correct w.r.t. this specification. Programming using primitive recursive schemes is really cumbersome and virtually nobody does this.

Compared to the above three steps approach, the trick which is used in this paper is to merge Items 1 and 2. Instead of showing that recursive functions are minimizations of primitive recursive functions, it is sufficient to show that *recursive functions are minimizations of Coq definable predicates*. From this point of view, it is possible to completely avoid the encoding/decoding phases from/to $\mathcal{N}$ but more importantly, we do not need a small-step semantics any more; we can replace it with a *decidable big-step semantics*: this avoids the implementation of a model of computation and thus, the proof of correctness of a compiler.

[2] It is difficult to use a word more accurate than "seems" because the relevant discussion in [2] is just a short outline of an approach, not a proof or an actual implementation.

Our mechanization proceeds in the following steps. We define an inductive predicate denoted $[f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x$ and called *cost aware big-step semantics*. It reads as: the recursive algorithm $f$ terminates on input $\boldsymbol{v}$ and outputs $x$ at cost $\alpha$. This relation is functional/deterministic in both $\alpha$ and $x$. We show the equivalence $[\![f]\!]\, \boldsymbol{v}\, x \iff \exists \alpha,\ [f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x$. We establish the central result of decidability of cost aware big-step semantics *when $\alpha$ is fixed:* for any $f$, $v$ and $\alpha$, either $x$ together with a proof of $[f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x$ can be computed (i.e. $\{x \mid [f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x\}$), or (an informative "or") a proof that no such $x$ exists can be computed (i.e. $\neg \exists x,\ [f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x$). These results are combined in the following way: from a proof of definedness $(\exists x, [\![f]\!]\, \boldsymbol{v}\, x)$, we deduce $\exists x \exists \alpha,\ [f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x$. Equivalently we get $\exists \alpha, \texttt{inhabited}\ \{x \mid [f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x\}$. By unbounded minimization of inhabited decidable predicates (see Sect. 3), we reify the proposition $\exists \alpha, \texttt{inhabited}\ \{x \mid [f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x\}$ into the predicate $\{\alpha \,\&\, \{x \mid [f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x\}\}$. Then we extract $\alpha$, $x$ and a proof that $[f; \boldsymbol{v}] \multimap [\alpha] \!\!\rangle\, x$, hence $[\![f]\!]\ \boldsymbol{v}\ x$, showing that the computed value $x$ is the output value of $f$ on input $\boldsymbol{v}$.

## 3 Reifying $\exists P$ into $\Sigma P$ for $P : \mathcal{N} \to \{\texttt{Prop}, \texttt{Type}\}$

In this section, we describe a way to reify non-informative inhabited decidable predicates of type $P : \mathcal{N} \to \texttt{Prop}$. So we show how to constructively build a value $n : \mathcal{N}$ and a proof term $t : P\, n$. We use an unbounded (but still well-founded) minimization algorithm whose termination is guaranteed by a proof of inhabitation $\exists n, P\, n$. The mechanization occurs in the file `nat_minimizer.v` which is nearly self-contained. In a way, this shows that Coq has unbounded minimization of inhabited and decidable predicates, whereas the theory of recursive functions has unbounded minimization of partial recursive functions. In Sect. 3.3, we also reify informative decidable predicates $P : \mathcal{N} \to \texttt{Type}$ that are inhabited, i.e. verifying $\exists n, \texttt{inhabited}\,(P\ n)$.

### 3.1 Existential Quantification in Coq

Let us recall the usual interpretation of the existential quantifiers that are available in Coq. In Type Theory, they are called $\Sigma$-types over a index type $X$:

- for $P : X \to \texttt{Prop}$, the expression $\exists x : X, P\ x$ (or $\texttt{ex}\, P$) is of type `Prop` and a term of that type is only a proof that there exists $x : X$ which satisfies $P\ x$. The witness $x$ need not be effective. It can be obtained by non-constructive means. For instance, the proof may use axioms in `Prop` such as the excluded middle (typically). We say that the predicate $\exists x : X, P\ x$ is *non-informative*;
- for $P : X \to \texttt{Prop}$, the expression $\{x : X \mid P\ x\}$ (or $\texttt{sig}\, P$) is of type `Set`/`Type` and a proof term for it is an (effective) term $x$ together with a proof of $P\ x$ ($x$ must be described by purely constructive methods). We say that the predicate $\{x : X \mid P\ x\}$ is *informative*;
- for $P : X \to \texttt{Type}$, the expression $\{x : X \,\&\, P\ x\}$ (or $\texttt{sigT}\, P$) is of type `Type`. It carries both an effective witness $x$ such that $P\ x$ is inhabited and an effective inhabitant of $P\, x$. The predicate $\{x : X \,\&\, P\ x\}$ is *fully informative*.

When the computational content of terms is extracted, the sub-terms of type `Prop` are pruned and their code does not impact the extracted terms: this property is called *proof irrelevance*. It implies that adding axioms in `Prop` will only allow to show more (termination) properties but it will not change the behaviour of terms. However, proof irrelevance is not preserved by adding axioms in `Type`.

The *Constructive Indefinite Description axiom* as stated in Coq standard library module `ChoiceFacts` can reify any non-informative predicate $\exists P$:

$$\forall (X : \texttt{Type})(P : X \to \texttt{Prop}), (\exists x : X, P\ x) \to \{x : X \mid P\ x\} \qquad \text{(CID)}$$

It provides an (axiomatic) transformation of $\exists P$ (i.e. $\exists x, P\ x$ in Coq) into $\Sigma P$ (i.e. $\{x \mid P\ x\}$ in Coq). The type $\forall X : \texttt{Type}, \texttt{inhabited}\ X \to X$ provides an equivalent definition of (CID) where $\texttt{inhabited} : \texttt{Type} \to \texttt{Prop}$ is the "hidding predicate" of the `Logic` module; see file `cid.v` and Sect. 3.3.

Assuming the axiom (CID) creates an "artificial" bridge between two separate worlds.[3] Some would even claim that such an axiom is at odds with the design philosophy of Coq: the default bridges that exist between the non-informative sort `Prop` and the informative sorts `Set`/`Type` were carefully introduced by Coq designers to be "constructively" safe; in particular, to ensure that extraction is proof irrelevant. Assuming (CID) would not be inconsistent with extraction but it would leave a hole in the extracted terms that make use of it. Moreover, assuming (CID), one can easily derive a proof of $\forall A\ B : \texttt{Prop}, A \vee B \to \{A\} + \{B\}$ and thus, a statement like $\forall x, \{P\ x\} + \{\neg P\ x\}$ cannot be interpreted as "$P$ is decidable" anymore. This is well explained in [3] together with the relations between (CID) and Hilbert's epsilon operator. You will also find a summary of the incompatibilities between (CID) and other features or axioms in Coq.

### 3.2 The Case of Predicates of Type $\mathcal{N} \to$ `Prop`

We describe a way to implement an instance of (CID) constructively but of course, that proof requires additional assumptions: we require that $P$ is a decidable predicate that ranges over $\mathcal{N}$ instead of an arbitrary type $X$. We do not extract the missing information $x$ but instead, we generate it using a well-founded algorithm that first transforms the non-informative inhabitation predicate $\exists x : \mathcal{N}, P\ x$ into a *termination certificate* for a well-founded minimization algorithm that sequentially enumerates natural numbers in *ascending order*.

Recall the definition of the non-informative *accessibility* predicate from the `Wf` module of the Coq standard library:

$$\begin{array}{l}\texttt{Inductive Acc}\ \{X : \texttt{Type}\}\ (R : X \to X \to \texttt{Prop})\ (x : X) :=\\ \quad \mid \texttt{Acc\_intro} : (\forall y : X, R\ y\ x \to \texttt{Acc}\ R\ y) \to \texttt{Acc}\ R\ x\end{array}$$

We write `Acc` $R$ instead of `Acc` $X\ R$ because the parameter $X$ is declared implicit.

[3] Of course this statement is of philosophical nature. We do not claim that assuming additional axiom is evil, but carelessly adding axioms is a recipe for inconsistencies.

We assume a predicate $P : \mathcal{N} \to \texttt{Prop}$ and we suppose that $P$ is decidable (in Coq) with a decision term $H_P$. We define a binary relation $R : \mathcal{N} \to \mathcal{N} \to \texttt{Prop}$ and we show the following results:

$$
\begin{array}{lll}
\texttt{Variables} & (P : \mathcal{N} \to \texttt{Prop})\ \big(H_P : \forall n : \mathcal{N},\ \{P\ n\} + \{\neg P\ n\}\big) & \\
\texttt{Let}\ R\ (n\ m : \mathcal{N}) & := & (n = 1 + m) \wedge \neg P\ m \\
\texttt{Let}\ \texttt{P\_Acc\_R} & : & \forall n : \mathcal{N},\ P\ n \to \texttt{Acc}\ R\ n \\
\texttt{Let}\ \texttt{Acc\_R\_dec} & : & \forall n : \mathcal{N},\ \texttt{Acc}\ R\ (1 + n) \to \texttt{Acc}\ R\ n \\
\texttt{Let}\ \texttt{Acc\_R\_zero} & : & \forall n : \mathcal{N},\ \texttt{Acc}\ R\ n \to \texttt{Acc}\ R\ 0 \\
\texttt{Let}\ \texttt{Acc\_P} & : & \forall n : \mathcal{N},\ \texttt{Acc}\ R\ n \to \{x : \mathcal{N} \mid P\ x\}
\end{array}
$$

which all have straightforward proofs except for `Acc_P`. That last one is done by induction on the accessibility predicate $\texttt{Acc}\ R\ n$. The proof term `Acc_P` uses the decision term $H_P$ to choose between stopping and moving on to the successor: it stops when $H_P\ n$ returns "true," i.e. $\texttt{left}\ T$ with $T : P\ n$; it loops on $1 + n$ when $H_P\ n$ returns "false," i.e. $\texttt{right}\ F$ with $F : \neg P\ n$. We analyse the term:

$$
\begin{array}{l}
\texttt{Let}\ \texttt{Acc\_inv}\ (n : \mathcal{N})\ (H_n : \texttt{Acc}\ R\ n) : \forall m, R\ m\ n \to \texttt{Acc}\ R\ m := \\
\quad \texttt{match}\ H_n\ \texttt{with}\ \texttt{Acc\_intro}\ \_\ H'_n \mapsto H'_n\ \texttt{end} \\
\texttt{Fixpoint}\ \texttt{Acc\_P}\ (n : \mathcal{N})\ (H_n : \texttt{Acc}\ R\ n) : \{x : \mathcal{N} \mid P\ x\} := \\
\quad \texttt{match}\ H_P\ n\ \texttt{with} \\
\qquad \mid \texttt{left}\ T \ \mapsto \texttt{exist}\ \_\ n\ T \\
\qquad \mid \texttt{right}\ F \mapsto \texttt{Acc\_P}\ (1 + n)\ \big(\texttt{Acc\_inv}\ \_\ H_n\ \_\ (\texttt{conj}\ \texttt{eq\_refl}\ F)\big) \\
\quad \texttt{end}.
\end{array}
$$

The recursion cannot be based on the argument $n$ because it would not be structurally well-founded in that case and the Coq type-checker would reject it. The recursion is based on the $\texttt{Acc}\ R\ n$ predicate. The definition is split in two parts to make it more readable; `Acc_inv` is from the module `Wf` of the standard library. The term `Acc_P` is a typical example of fixpoint by induction over an ad hoc predicate (see [2] or *the Coq'Art* [1] p. 428). The `Fix_F` fixpoint operator of the `Wf` module of the Coq standard library is defined this way as well. The *cover-induction principle* as defined in [4] uses a similar idea.

As a consequence, we can reify decidable and inhabited predicates over $\mathcal{N}$:

$$
\begin{array}{l}
\texttt{Theorem}\ \texttt{nat\_reify}\ (P : \mathcal{N} \to \texttt{Prop}) : \\
\qquad \big(\forall n : \mathcal{N}, \{P\ n\} + \{\neg P\ n\}\big) \to \big(\exists n : \mathcal{N}, P\ n\big) \to \big\{n : \mathcal{N} \mid P\ n\big\}
\end{array}
$$

The proof is now simple: using `P_Acc_R` and `Acc_R_zero`, from $\exists n, P\,n$ we deduce $\texttt{Acc}\ R\ 0$, and thus $\{x : \mathcal{N} \mid P\ x\}$ using `Acc_P`.

Considering this somewhat unexpected result, maybe some further clarifications about the proof of `nat_reify` are mandatory at this stage. The witness $n$ which is contained in the hypothesis $\exists n, P\ n$ of sort `Prop` is not informative and thus cannot be extracted to build a term of sort `Type`. As this remarks seems contradictory with what we show, we insist on the fact that we do not extract the witness $n$ contained in the hypothesis by inspection of its term. Instead, we compute the minimum value $m$ which satisfies $P\,m$ by testing all cases in

sequence: $P\,0$ ?, $P\,1$ ?, ... until we reach the first value $m$ which satisfies $P\,m$ (the decidability of $P$ is required for that). To ensure that such a computation is well-founded, we use the non-informative witness $n$ contained in $\exists n, P\,n$ as a *bound on the search space*; but a bound in sort `Prop`: we encode $n$ into the accessibility predicate $A_n : \texttt{Acc}\ R\ 0$ which is then used as a certificate for the well-foundedness of the computation of $\texttt{Acc\_P}\ 0\ A_n$.

### 3.3 Reification of Predicates of Type $\mathcal{N} \to$ `Type`

We now generalize the previous result `nat_reify` to predicates in $\mathcal{N} \to$ `Type` instead of just $\mathcal{N} \to$ `Prop`. But we first need to introduce two predicates:

$$\texttt{Inductive}\quad \texttt{inhabited}\ (P : \texttt{Type}) : \texttt{Prop} := \texttt{inhabits} : P \to \texttt{inhabited}\ P$$
$$\texttt{Definition}\quad \texttt{decidable\_t}\ (P : \texttt{Type}) : \texttt{Type} := P + P \to \texttt{False}$$

where `inhabited` is from the standard library (module `Logic`) and `decidable_t` is an informative version of the `decidable` predicate of the `Decidable` module of the standard library. Their intuitive meaning is the following:

- `inhabited` $P$ hides the information of the witness of $P$. Whereas a term of type $P$ is a witness that $P$ is inhabited, a term of type `inhabited` $P$ hides the witness by the use of the non-informative constructor `inhabits`;
- `decidable_t` $P$ means that either a term of type $P$ is given or a proof that $P$ is void is given. The predicate is informative and contains a Boolean choice (represented by the $+$) which tells whether $P$ is inhabited or not. But it may also contain an effective witness that $P$ is inhabited.

We can now lift the theorem `nat_reify` that operates on $\mathcal{N} \to$ `Prop` to informative predicates of type $\mathcal{N} \to$ `Type` in the following way:

$$\texttt{Theorem nat\_reify\_t}\ \ (P : \mathcal{N} \to \texttt{Type}) :$$
$$\big(\forall n, \texttt{decidable\_t}\,(P\ n)\big) \to \big(\exists n, \texttt{inhabited}\,(P\ n)\big) \to \big\{n : \mathcal{N}\ \&\ P\ n\big\}$$

The proof is only a slight variation from the $\mathcal{N} \to$ `Prop` case. Notice that the result type $\{n : \mathcal{N}\ \&\ P\ n\}$ contains the reified value $n$ for which $P\,n$ is inhabited, but it also contains the effective witness that $P\,n$ is not void. On the contrary, in the hypothesis $\exists n, \texttt{inhabited}\,(P\,n)$ neither $n$ nor the witness that $P\,n$ is inhabited have to be provided by effective means.

## 4 Dependent Types for Recursive Algorithms

So far, we have only encountered datatypes which originate in the Coq standard library, and that are imported by default when loading Coq, most notably $\mathcal{N}$ which is a least solution of the fixpoint equation $\mathcal{N} \equiv \{\texttt{0}\} + \{\texttt{S}\ n \mid n : \mathcal{N}\}$. We will need the type of vectors `VectorDef.t` and the type of positions `Fin.t` that also belong to the standard library module `Vector`. However, the standard library only contains a small fraction of the results that we use for these

datatypes. Moreover, *the implementation of some functions of the* `Vector` *module is incompatible with how we intend to use them.* Typically, the definition of `VectorDef.nth` which selects a component of a vector by its position does not type-check in our succinct definition of the upcoming `recalg_rect` recursor: the definition of `VectorDef.nth` makes Coq unable to certify the structural decrease of recursive sub-calls which is mandatory for `Fixpoint` definitions. As a consequence, we use our own vectors and positions libraries. This represents little overhead compared to extending the standard libraries in the `Vector` module.

We define three types that depend on a parameter $k : \mathcal{N}$ representing an arity. First the type of *positions*

$$\mathtt{pos}\,0 \equiv \emptyset \qquad \mathtt{pos}(1+k) \equiv \{\mathtt{fst}\} + \{\mathtt{nxt}\,p \mid p : \mathtt{pos}\,k\}$$

which is isomorphic to $\mathtt{pos}\,k \equiv \{i : \mathcal{N} \mid i < k\}$ but avoids carrying the proof term $i < k$. The library `pos.v` contains the inductive definitions of the type $\mathtt{pos}\,k$ and the tools to manipulate positions smoothly: an inversion tactic `pos_inv`, maps $\mathtt{pos2nat} : \mathtt{pos}\,k \to \mathcal{N}$ and $\mathtt{nat2pos} : \forall i, i < k \to \mathtt{pos}\,k$, etc. To shorten the notations in this paper, $\overline{p}$ denotes `pos2nat` $p$, the natural number below $k$ which corresponds to $p$.

Positions of $\mathtt{pos}\,k$ mainly serve as coordinates for accessing the components of *vectors of arity* $k$

$$X^0 \equiv \{\mathtt{vec\_nil}\} \qquad X^{1+k} \equiv X \times X^k$$

where $X^k$ is a compact notation for `vec` $X$ $k$. The type is polymorphic in $X$ and dependent on $k : \mathcal{N}$. We will write terms of type $X^k$ in a bold font like with $\boldsymbol{v}$ or $\boldsymbol{w}$ to remind the reader that these are vectors. Given a position $p : \mathtt{pos}\,k$ and a vector $\boldsymbol{v} : X^k$, we write $\boldsymbol{v}_p : X$ for the $p$-th component of $\boldsymbol{v}$, a short-cut for `vec_pos` $\boldsymbol{v}$ $p$. `vec_pos` is obtained from the "correspondence" $X^k \equiv \mathtt{pos}\,k \to X$. Notice however that the type $X^k$ enjoys an extensional equality (i.e. $\boldsymbol{v} = \boldsymbol{w}$ whenever $\boldsymbol{v}_p = \boldsymbol{w}_p$ holds for any $p : \mathtt{pos}\,k$) whereas the function type $\mathtt{pos}\,k \to X$ does not. The file `vec.v` contains the inductive definition of the type of vectors together with the tools to smoothly manipulate vectors and their components where coordinates can either be positions of type $\mathtt{pos}\,k$ or natural number $i : \mathcal{N}$ satisfying $i < k$. The constructors are $\mathtt{vec\_nil} : X^0$ and $\mathtt{vec\_cons} : X \to X^k \to X^{1+k}$ and `vec_cons` $x$ $\boldsymbol{v}$ is denoted $x\#\boldsymbol{v}$ here. The converse operations are $\mathtt{vec\_head} : X^{1+k} \to X$ and $\mathtt{vec\_tail} : X^{1+k} \to X^k$.

With positions and (polymorphic) vectors, we can now introduce the inductive type of *recursive algorithms of arity* $k$ denoted by $\mathcal{A}_k$ which is defined by the rules of Fig. 1 and implemented in the file `recalg.v`. The notation $\mathcal{A}_k$ is a short-cut for `recalg` $k$. Notice that $\mathcal{A}_k$ is a dependent type (of sort `Set`). It is the least type which contains *constants* of arity 0, *zero* and *succ* of arity 1, *projections* at every arity $k$ for each possible coordinate, and which is closed under the *composition*, *primitive recursion* and *unbounded minimization* schemes. $\mathcal{A}_k$ itself does not carry the semantics of those recursive algorithms: it corresponds to the source code. We will give a meaning/semantics to those recursive algorithms in Sect. 5 so that they correspond to the usual notion of recursive functions.

$$\frac{n : \mathcal{N}}{\texttt{cst}_n : \mathcal{A}_0} \qquad \frac{}{\texttt{zero} : \mathcal{A}_1} \qquad \frac{}{\texttt{succ} : \mathcal{A}_1} \qquad \frac{p : \texttt{pos}\, k}{\texttt{proj}_p : \mathcal{A}_k}$$

$$\frac{f : \mathcal{A}_k \quad \boldsymbol{g} : \mathcal{A}_i^k}{\texttt{comp}\, f\, \boldsymbol{g} : \mathcal{A}_i} \qquad \frac{f : \mathcal{A}_k \quad g : \mathcal{A}_{2+k}}{\texttt{rec}\, f\, g : \mathcal{A}_{1+k}} \qquad \frac{f : \mathcal{A}_{1+k}}{\texttt{min}\, f : \mathcal{A}_k}$$

**Fig. 1.** The type $\mathcal{A}_k$ of recursive algorithms of arity $k$.

To be able to compute with or prove properties of terms of type $\mathcal{A}_k$, we implement a general fully dependent recursion scheme `recalg_rect` described in the file recalg.v. This principle is not automatically generated by Coq because of the nested induction between the types $\mathcal{A}_k$ and `vec _ k` which occurs in the constructor `comp` $f\, \boldsymbol{g}$. The definition of `recalg_rect` looks simple but it only works well because `vec_pos` was carefully designed to allow the Coq type-checker to detect nested recursive calls as structurally simpler: using the "equivalent" `VectorDef.nth` instead of `vec_pos` prohibits successful type-checking. We also show the injectivity of the constructors of the type $\mathcal{A}_k$. Some require the use of the `Eqdep_dec` module of the standard library because of the dependently typed context. For example, the statement of the injectivity of the constructor `comp` $f\, \boldsymbol{g}$ involves type castings `eq_rect` (or heterogenous equality):

$$\texttt{Fact ra\_comp\_inj}\ k\ k'\ i\ (f : \mathcal{A}_k)\ (f' : \mathcal{A}_{k'})\ (\boldsymbol{g} : \mathcal{A}_i^k)\ (\boldsymbol{g}' : \mathcal{A}_i^{k'}) :$$
$$\texttt{comp}\, f\, \boldsymbol{g} = \texttt{comp}\, f'\, \boldsymbol{g}' \to \exists e : k = k', \wedge \begin{cases} \texttt{eq\_rect}\ \_\ \_\ f\ \_\ e = f' \\ \texttt{eq\_rect}\ \_\ \_\ \boldsymbol{g}\ \_\ e = \boldsymbol{g}' \end{cases}$$

## 5 A Decidable Semantics for Recursive Algorithms

In this section, we define three equivalent semantics for recursive algorithms. First the standard *relational semantics* defined by recursion on $f : \mathcal{A}_k$, then an equivalent *big-step semantics* defined by a set of inductive rules. Those two semantics cannot be decided. Then we define a refinement of big-step semantics by annotating it with a cost. By constraining the cost, we obtain a decidable semantics for recursive algorithms $\mathcal{A}_k$.

### 5.1 Relational and Big-Step Semantics

We define relational semantics $[\![f : \mathcal{A}_k]\!] : \mathcal{N}^k \to \mathcal{N} \to \texttt{Prop}$ of recursive algorithms by structural recursion on $f : \mathcal{A}_k$ so as to satisfy the fixpoint equations of Fig. 2 where $[\![f]\!]$ is a notation for `ra_rel` $f$; the fixpoint equations `ra_rel_fix_`$*$ are proved in the file ra_rel.v. Without preparation, such a definition could be quite technical because of the nested recursion between the type $\mathcal{A}_k$ and the type `vec` $\mathcal{A}_i$ $k$ of the parameter $\boldsymbol{g}$ in the constructor `comp` $f\, \boldsymbol{g}$. Using our general recursion principle `recalg_rect`, the code is straightforward; but see the remark

$$
\begin{array}{rcl@{\qquad}rcl}
[\![\mathtt{cst}_n]\!]\ \_\ x & \iff & n = x & [\![\mathtt{zero}]\!]\ \_\ x & \iff & 0 = x \\
[\![\mathtt{succ}]\!]\ \boldsymbol{v}\ x & \iff & 1 + \boldsymbol{v}_{\mathtt{fst}} = x & [\![\mathtt{proj}_p]\!]\ \boldsymbol{v}\ x & \iff & \boldsymbol{v}_p = x \\
[\![\mathtt{comp}\ f\ g]\!]\ \boldsymbol{v}\ x & \iff & \exists \boldsymbol{w}, [\![f]\!]\ \boldsymbol{w}\ x \wedge \forall p, [\![\boldsymbol{g}_p]\!]\ \boldsymbol{v}\ \boldsymbol{w}_p & & & \\
[\![\mathtt{rec}\ f\ g]\!]\ (0\#\boldsymbol{v})\ x & \iff & [\![f]\!]\ \boldsymbol{v}\ x & & & \\
[\![\mathtt{rec}\ f\ g]\!]\ (1+n\#\boldsymbol{v})\ x & \iff & \exists y, [\![\mathtt{rec}\ f\ g]\!]\ (n\#\boldsymbol{v})\ y \wedge [\![g]\!]\ (n\#y\#\boldsymbol{v})\ x & & & \\
[\![\mathtt{min}\ f]\!]\ \boldsymbol{v}\ x & \iff & \exists \boldsymbol{w}, [\![f]\!]\ (x\#\boldsymbol{v})\ 0 \wedge \forall p : \mathtt{pos}\ x, [\![f]\!]\ (\overline{p}\#\boldsymbol{v})\ (1+\boldsymbol{w}_p) & & &
\end{array}
$$

**Fig. 2.** Relational semantics `ra_rel` for recursive algorithms of $\mathcal{A}_k$.

about `recalg_rect` in Sect. 4. We explicitly mention the type $p : \mathtt{pos}\ x$ in the definition of $[\![\mathtt{min}\ f]\!]$ because it is the only type which does not depend on the type of $f$: the dependent parameter $x$ is the result of the computation.

The big-step semantics for recursive algorithms in $\mathcal{A}_k$ is an inductive relation of type $\mathtt{ra\_bs} : \forall k, \mathcal{A}_k \to \mathcal{N}^k \to \mathcal{N} \to \mathtt{Prop}$ and we denote $[f; \boldsymbol{v}] \rightsquigarrow x$ for $(\mathtt{ra\_bs}\ k\ f\ \boldsymbol{v}\ x)$; the parameter $k$ is implicit in the notation. $[f; \boldsymbol{v}] \rightsquigarrow x$ intuitively means that the computation of $f$ starting from input $\boldsymbol{v}$ yields the result $x$. We define big-step semantics in file `ra_bs.v` by the inductive rules of Fig. 3. We point out that the rule corresponding to $[\mathtt{min}\ f; \boldsymbol{v}] \rightsquigarrow x$ is of unbounded arity but still finitary because $\mathtt{pos}\ x$ is a finite type. These rules are similar to those used to define the semantics of Partial Recursive Functions in [13] except that thanks to our dependent typing, we do not need to specify *well-formedness conditions*. In `ra_sem_eq.v`, we show that big-step semantics is equivalent to relational semantics:

$$\mathtt{Theorem}\ \ \mathtt{ra\_bs\_correct}\ k\ (f : \mathcal{A}_k)\ (\boldsymbol{v} : \mathcal{N}^k)\ x : \quad [\![f]\!]\ \boldsymbol{v}\ x \iff [f; \boldsymbol{v}] \rightsquigarrow x$$

However big-step semantics has the advantage of being defined by a set of inductive rules instead of being defined by recursion on $f : \mathcal{A}_k$.

Relational and big-step semantics are not recursive/computable relations: this is an instance of the *Halting problem*. As such, these relations cannot be implemented by a Coq evaluation function $\mathtt{ra\_rel\_eval} : \mathcal{A}_k \to \mathcal{N}^k \to \mathtt{option}\ \mathcal{N}$ satisfying $\mathtt{ra\_rel\_eval}\ f\ \boldsymbol{v} = \mathtt{Some}\ x \iff [\![f]\!]\ \boldsymbol{v}\ x$ for any $f$, $\boldsymbol{v}$ and $x$. Indeed, when it is axiom free, Coq has normalisation which implies that the functions that can be defined in it are total recursive at the meta-theoretical level. Nevertheless big-step semantics as presented in Fig. 3 is an intermediate step towards a decidable semantics for $\mathcal{A}_k$.

### 5.2 Cost Aware Big-Step Semantics

The cost aware big-step semantics for recursive algorithms in $\mathcal{A}_k$ is defined as an inductive predicate of type $\mathtt{ra\_ca} : \forall k, \mathcal{A}_k \to \mathcal{N}^k \to \mathcal{N} \to \mathcal{N} \to \mathtt{Prop}$. We denote $(\mathtt{ra\_ca}\ k\ f\ \boldsymbol{v}\ \alpha\ x)$ by $[f; \boldsymbol{v}] \multimap\!\langle \alpha \rangle\, x$ where the argument $k$ is implicit in the notation. $[f; \boldsymbol{v}] \multimap\!\langle \alpha \rangle\, x$ intuitively means that the computation of $f$ on input $\boldsymbol{v}$ yields the result $x$ and costs $\alpha$. We define the predicate `ra_ca` in file `ra_ca.v` by the rules of Fig. 4. It is interesting to compare these rules with those of conventional big-step semantics `ra_bs` of Fig. 3. The very simple but nonetheless powerful idea

$$\frac{}{[\mathtt{cst}_n; \boldsymbol{v}] \rightsquigarrow n} \quad \frac{}{[\mathtt{zero}; \boldsymbol{v}] \rightsquigarrow 0} \quad \frac{}{[\mathtt{succ}; \boldsymbol{v}] \rightsquigarrow 1 + \boldsymbol{v}_{\mathtt{fst}}} \quad \frac{}{[\mathtt{proj}_p; \boldsymbol{v}] \rightsquigarrow \boldsymbol{v}_p}$$

$$\frac{[f; \boldsymbol{v}] \rightsquigarrow x}{[\mathtt{rec}\, f\, g; 0 \# \boldsymbol{v}] \rightsquigarrow x} \quad \frac{[\mathtt{rec}\, f\, g; n \# \boldsymbol{v}] \rightsquigarrow y \quad [g; n \# y \# \boldsymbol{v}] \rightsquigarrow x}{[\mathtt{rec}\, f\, g; 1 + n \# \boldsymbol{v}] \rightsquigarrow x}$$

$$\frac{[f; \boldsymbol{w}] \rightsquigarrow x \quad \forall p, [\boldsymbol{g}_p; \boldsymbol{v}] \rightsquigarrow \boldsymbol{w}_p}{[\mathtt{comp}\, f\, \boldsymbol{g}; \boldsymbol{v}] \rightsquigarrow x} \quad \frac{[f; x \# \boldsymbol{v}] \rightsquigarrow 0 \quad \forall p : \mathtt{pos}\, x, [f; \overline{p} \# \boldsymbol{v}] \rightsquigarrow 1 + \boldsymbol{w}_p}{[\mathtt{min}\, f; \boldsymbol{v}] \rightsquigarrow x}$$

**Fig. 3.** Big-step semantics `ra_bs` for recursive algorithms of $\mathcal{A}_k$.

$$\frac{}{[\mathtt{cst}_n; \boldsymbol{v}] \dashv\!\![1]\!\!\rangle\, n} \quad \frac{}{[\mathtt{zero}; \boldsymbol{v}] \dashv\!\![1]\!\!\rangle\, 0} \quad \frac{}{[\mathtt{succ}; \boldsymbol{v}] \dashv\!\![1]\!\!\rangle\, 1 + \boldsymbol{v}_{\mathtt{fst}}} \quad \frac{}{[\mathtt{proj}_p; \boldsymbol{v}] \dashv\!\![1]\!\!\rangle\, \boldsymbol{v}_p}$$

$$\frac{[f; \boldsymbol{v}] \dashv\!\![\alpha]\!\!\rangle\, x}{[\mathtt{rec}\, f\, g; 0 \# \boldsymbol{v}] \dashv\!\![1 + \alpha]\!\!\rangle\, x} \quad \frac{[\mathtt{rec}\, f\, g; n \# \boldsymbol{v}] \dashv\!\![\alpha]\!\!\rangle\, y \quad [g; n \# y \# \boldsymbol{v}] \dashv\!\![\beta]\!\!\rangle\, x}{[\mathtt{rec}\, f\, g; 1 + n \# \boldsymbol{v}] \dashv\!\![1 + \alpha + \beta]\!\!\rangle\, x}$$

$$\frac{[f; \boldsymbol{w}] \dashv\!\![\alpha]\!\!\rangle\, x \quad \forall p, [\boldsymbol{g}_p; \boldsymbol{v}] \dashv\!\![\boldsymbol{\beta}_p]\!\!\rangle\, \boldsymbol{w}_p}{[\mathtt{comp}\, f\, \boldsymbol{g}; \boldsymbol{v}] \dashv\!\![1 + \alpha + \Sigma\boldsymbol{\beta}]\!\!\rangle\, x} \quad \frac{[f; x \# \boldsymbol{v}] \dashv\!\![\alpha]\!\!\rangle\, 0 \quad \forall p : \mathtt{pos}\, x, [f; \overline{p} \# \boldsymbol{v}] \dashv\!\![\boldsymbol{\beta}_p]\!\!\rangle\, 1 + \boldsymbol{w}_p}{[\mathtt{min}\, f; \boldsymbol{v}] \dashv\!\![1 + \alpha + \Sigma\boldsymbol{\beta}]\!\!\rangle\, x}$$

**Fig. 4.** Cost aware big-step semantic `ra_ca` for recursive algorithms of $\mathcal{A}_k$.

to get decidability is to decorate big-step semantics with a cost and to constrain computations by a cost that must be exactly matched. This is how we realize the principle of our proof that Coq contains total recursive functions: we avoid a small-step semantics (Kleene's T predicate) and replace it with a big-step semantics for recursive algorithm that is nevertheless decidable.

We show the equivalence of relational and cost aware big-step semantics

$$\begin{array}{l}\mathtt{Theorem}\ \ \mathtt{ra\_ca\_correct}\ (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (\boldsymbol{v} : \mathcal{N}^k)\ (x : \mathcal{N}) : \\ \qquad\qquad [\![f]\!]\ \boldsymbol{v}\ x \iff \exists \alpha : \mathcal{N}, [f; \boldsymbol{v}] \dashv\!\![\alpha]\!\!\rangle\, x\end{array}$$

in file `ra_sem_eq.v`. The proof is circular in style: `ra_ca` implies `ra_bs` implies `ra_rel` implies ∃`ra_ca` and all these three implications are proved by induction on the obvious inductive parameter. Do not feel puzzled by a statement of equivalence between a decidable and an undecidable semantics, because it is the quantifier $\exists\alpha$ in `ra_ca_correct` which brings undecidability.

Inversion lemmas named `ra_ca_*_inv` are essential tools to prove the high-level properties of Sect. 5.3. They allow case analysis on the last step of an inductive term depending on the shape of the conclusion. Here is the inversion lemma of one rule:

$$\begin{array}{l}\mathtt{Lemma}\ \ \mathtt{ra\_ca\_rec\_S\_inv}\ (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (g : \mathcal{A}_{2+k})\ (\boldsymbol{v} : \mathcal{N}^k)\ (n\ \gamma\ x : \mathcal{N}) : \\ [\mathtt{rec}\, f\, g; 1 + n \# \boldsymbol{v}] \dashv\!\![\gamma]\!\!\rangle\, x \to \exists y\ \alpha\ \beta, \wedge \begin{cases} \gamma = 1 + \alpha + \beta \\ [\mathtt{rec}\, f\, g; n \# \boldsymbol{v}] \dashv\!\![\alpha]\!\!\rangle\, y \\ [g; n \# y \# \boldsymbol{v}] \dashv\!\![\beta]\!\!\rangle\, x \end{cases}\end{array}$$

Such results could be difficult to establish if improperly prepared. In our opinion, the easiest way to prove it is to implement a global inversion lemma that encompasses the whole set of rules of Fig. 4. Then a lemma like `ra_ca_rec_S_inv` can be proved by applying the global inversion lemma and discriminate between incompatible constructors of type $\mathcal{A}_k$ (in most cases) or use injectivity of thoses constructors (in one case). The global inversion lemma is quite complicated to write because of dependent types. It would fill nearly half of this page (see lemma `ra_ca_inv` in file `ra_ca.v`). However it is actually trivial to prove, a "reversed" situation which is rare enough to be noticed.

### 5.3 Properties of Cost Aware Big-Step Semantics

The annotation of cost in the rules of Fig. 4 satisfies the following paradigm: the cost of a compound computation is greater than the sum of the costs of its sub-computations. Hence, we can derive that no computation is free of charge:

$$\texttt{Theorem ra\_ca\_cost } k\ (f : \mathcal{A}_k)\ (\boldsymbol{v} : \mathcal{N}^k)\ (\alpha\ x : \mathcal{N}) : \quad [f; \boldsymbol{v}] \dashv\!\lfloor \alpha \rfloor\!\rangle\, x \to 0 < \alpha$$

The proof is by immediate case analysis on $[f; \boldsymbol{v}] \dashv\!\lfloor \alpha \rfloor\!\rangle\, x$. The cost and results given by cost aware big-step semantics are unique (provided they exist)

$$\texttt{Theorem ra\_ca\_fun } (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (\boldsymbol{v} : \mathcal{N}^k)\ (\alpha\, \beta\, x\, y : \mathcal{N}) : \\ [f; \boldsymbol{v}] \dashv\!\lfloor \alpha \rfloor\!\rangle\, x \to [f; \boldsymbol{v}] \dashv\!\lfloor \beta \rfloor\!\rangle\, y \to \alpha = \beta \wedge x = y$$

The proof is by induction on $[f; \boldsymbol{v}] \dashv\!\lfloor \alpha \rfloor\!\rangle\, x$ together with inversion lemmas `ra_ca_*_inv` to decompose $[f; \boldsymbol{v}] \dashv\!\lfloor \beta \rfloor\!\rangle\, y$. Inversion lemmas are the central ingredient of this proof.

Now the key result: cost aware big-step semantics is decidable (in sort `Type`, see Sect. 3.3) *when the cost is fixed*

$$\texttt{Theorem ra\_ca\_decidable\_t } (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (\boldsymbol{v} : \mathcal{N}^k)\ (\alpha : \mathcal{N}) : \\ \texttt{decidable\_t}\ \{x \mid [f; \boldsymbol{v}] \dashv\!\lfloor \alpha \rfloor\!\rangle\, x\}$$

Its proof is the most complicated of our whole development. It proceeds by induction on $f : \mathcal{A}_k$ and uses inversion lemmas `ra_ca_*_inv`, functionality `ra_ca_fun` as well as a small *decidability library* to lift decidability arguments over (finitely) quantified statements. The central constituents of that library are:

$$\texttt{Lemma decidable\_t\_bounded } (P : \mathcal{N} \to \texttt{Type}) : \\ \big(\forall n : \mathcal{N},\ \texttt{decidable\_t}\,(P\ n)\big) \\ \to \forall n : \mathcal{N},\ \texttt{decidable\_t}\,\{i : \mathcal{N} \,\&\, i < n \times P\ i\}$$

$$\texttt{Lemma vec\_sum\_decide\_t } (n : \mathcal{N})\ (\boldsymbol{P} : (\mathcal{N} \to \texttt{Type})^n) : \\ \big(\forall (p : \texttt{pos}\, n)\ (i : \mathcal{N}),\ \texttt{decidable\_t}\,(\boldsymbol{P}_p\ i)\big) \\ \to \forall m : \mathcal{N},\ \texttt{decidable\_t}\,\{\boldsymbol{v} : \mathcal{N}^n \,\&\, \Sigma \boldsymbol{v} = m \times \forall p, \boldsymbol{P}_p\ \boldsymbol{v}_p\}$$

$$\texttt{Lemma vec\_sum\_unbounded\_decide\_t } (P : \mathcal{N} \to \mathcal{N} \to \texttt{Type}) :$$
$$\big(\forall n\ i : \mathcal{N},\ \texttt{decidable\_t}\,(P\ n\ i)\big)$$
$$\to \big(\forall n : \mathcal{N},\ P\ n\ 0 \to \texttt{False}\big)$$
$$\to \forall m : \mathcal{N},\ \texttt{decidable\_t}\,\big\{n : \mathcal{N}\ \&\ \{\boldsymbol{q} : \mathcal{N}^n\ \&\ \Sigma \boldsymbol{q} = m \times \forall p, P\ \overline{p}\ \boldsymbol{q}_p\}\big\}$$

Some comments about the intuitive meaning of such results could be useful. Recall that decidability has to be understood over `Type` (as opposed to `Prop`):

- `decidable_t_bounded` states that whenever $P\,n$ is decidable for any $n$, then given a bound $m$, it is decidable whether there exists $i < m$ such that $P\,i$ holds. Hence bounded existential quantification inherits decidability;
- `vec_sum_decide_t` states that if $\boldsymbol{P}$ is a $\texttt{pos}\,n \times \mathcal{N}$ indexed family of decidable predicates, then it is decidable whether there exists vector $\boldsymbol{v} : \mathcal{N}^n$ (of length $n$) which satisfies $\boldsymbol{P}_p\ \boldsymbol{v}_p$ for each of its components (indexed by $p : \texttt{pos}\,n$), and such that the sum of the components of $\boldsymbol{v}$ is a fixed value $m$. This express the decidability of some kind of universal quantification bounded by the length of a vector;
- `vec_sum_unbounded_decide_t` states that if $P$ is a $\mathcal{N} \times \mathcal{N}$ indexed family of decidable predicates such that $P\ \_\ 0$ is never satisfied, then it is decidable whether there exists a vector $\boldsymbol{q}$ of arbitrary length which satisfies $P$ at every component and such that the sum of those components is a fixed value $m$. This is a variant of `vec_sum_decide_t` but for unbounded vector length, only the sum of the components acts as a bound.

Once `ra_ca_decidable_t` is established, we combine it with `ra_ca_fun` to easily define a bounded computation function for recursive algorithms, as is done for instance at the end of file `ra_ca_props.v`:

$$\texttt{Definition}\quad \texttt{ra\_ca\_eval}\ (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (\boldsymbol{v} : \mathcal{N}^k)\ (\alpha : \mathcal{N}) : \texttt{option}\ \mathcal{N}$$

$$\texttt{Proposition}\quad \texttt{ra\_ca\_eval\_prop}\ (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (\boldsymbol{v} : \mathcal{N}^k)\ (\alpha\ x : \mathcal{N}) :$$
$$[f; \boldsymbol{v}] \dashv\!\lfloor \alpha \rfloor\!\rangle\ x \iff \texttt{ra\_ca\_eval}\ f\ \boldsymbol{v}\ \alpha = \texttt{Some}\ x$$

Notice that the function `ra_ca_eval` could be proved primitive recursive with proper encoding of $\mathcal{A}_k$ into $\mathcal{N}$ but the whole point of this work is to avoid having to program with primitive recursive schemes.

## 6 The Totality of Coq

In this section, we conclude our proof that Coq contains all the recursive functions for which totality can be established in Coq. We assume an arity $k : \mathcal{N}$ and a recursive algorithm $f : \mathcal{A}_k$ which is supposed to be total:

$$\texttt{Variables}\ \ (k : \mathcal{N})\ \ (f : \mathcal{A}_k)\ \ \big(H_f : \forall \boldsymbol{v} : \mathcal{N}^k, \exists x : \mathcal{N},\ [\![f]\!]\ \boldsymbol{v}\ x\big)$$

Mimicking Coq sectioning mechanism, these assumptions hold for the rest of the current section. We first show that given an input vector $\boldsymbol{v} : \mathcal{N}^k$, both a cost $\alpha : \mathcal{N}$ and a result $x : \mathcal{N}$ can be computed constructively:

$$\texttt{Let coq\_f}\ (\boldsymbol{v} : \mathcal{N}^k) : \big\{\alpha : \mathcal{N}\ \&\ \{x : \mathcal{N} \mid [f; \boldsymbol{v}] \dashv\!\lfloor \alpha \rfloor\!\rangle\ x\}\big\}$$

The proof uses unbounded minimization as implemented in `nat_reify_t` to find a cost $\alpha$ such that $\{x : \mathcal{N} \mid [f; \boldsymbol{v}] \dashv\!\lfloor \alpha \rfloor\!\rangle\, x\}$ is an inhabited type. This can be decided for each possible cost thanks to `ra_ca_decidable_t`. Recall that `nat_reify_t` tries 0, then 1, then 2, etc. until it finds the one which is guaranteed to exist. The warranty is provided by a combination of $H_f$ and `ra_ca_correct`.

To obtain the predicate $t : \mathcal{N}^k \to \mathcal{N}$ that realizes $[\![f]\!]$, we simply permute $x$ and $\alpha$ in `coq_f` $\boldsymbol{v}$. We define $t := \boldsymbol{v} \mapsto$ `proj1_sig(projT2(coq_f` $\boldsymbol{v}$`))`. Using `projT1(coq_f` $\boldsymbol{v}$`)`, `proj2_sig(projT2(coq_f` $\boldsymbol{v}$`))` and `ra_ca_correct`, it is trivial to show that $t\ \boldsymbol{v}$ satisfies $[\![f]\!]\ \boldsymbol{v}\ (t\ \boldsymbol{v})$. Hence, closing the section and discharging the local assumptions, we deduce the totality theorem.

```
Theorem Coq_is_total (k : 𝒩) (f : 𝒜_k) :
  (∀ v : 𝒩^k, ∃ x : 𝒩, ⟦f⟧ v x) → {t : 𝒩^k → 𝒩 | ∀ v : 𝒩^k, ⟦f⟧ v (t v)}
```

## 7 Discussion: Other Approaches, Church Thesis

Comparing our method with the approach based on Kleene's normal form theorem (Sect. 2), we remark that the introduction of small-step semantics would only be used to measure the length (or cost) of computations. Since there is at most one computation from a given input in deterministic models of computation, any computation can be recovered from its number of steps *by primitive recursive means*. Hence the idea of short-cutting small-step semantics by a cost.

It is not surprising that the Kleene's normal form approach was only suggested in [2]. Mechanizing a Turing complete model of computation is bound to be a lengthy development. Mainly because translating between elementary models of computation resembles writing programs in assembly language that you moreover have to specify and prove correct. And unsurprisingly, such developments are relatively rare and recent, with the notable exception of [13] which formalizes computability notions in Coq. $\mu$-recursive functions are not dependently typed in [13] (so there is a well-formedness predicate) and they are not compiled into a model of execution. In [12] however, the same author presents a compiler from $\mu$-recursive functions to Unlimited Register Machines, proved correct in HOL. Turing machines, Abacus machines and $\mu$-recursive functions are implemented in [11] with the aim of been able to characterize decidability in HOL. The development in [8] approaches computability in HOL4 through $\lambda$-calculus also with the aim at the mechanization of computability arguments. We recently published online a constructive implementation in (axiom-free) Coq [7] of an significant portion of Krivine's textbook [6] on $\lambda$-calculus, including a translation from $\mu$-recursive functions to $\lambda$-terms with dependent types in Coq. Actually, this gave us a first mechanized proof that Coq contained any total recursive function by using leftmost $\beta$-reduction strategy to compute normal forms. But it requires the introduction of intersection type systems, a development of more than 25 000 lines of code.

Now, what about a characterization of the functions of type $\mathcal{N}\to\mathcal{N}$ definable in Coq? Or else, is such a converse statement of (CiT)

$$\forall(k:\mathcal{N})\ (g:\mathcal{N}^k\to\mathcal{N}), \exists f:\mathcal{A}_k, \forall \boldsymbol{v}:\mathcal{N}^k,\ [\![f]\!]\ \boldsymbol{v}\ (g\ \boldsymbol{v}) \qquad \text{(ChT)}$$

provable in Coq? It is not too difficult to see that (ChT) does not hold in a model of Coq where function types contain the full set of set theoretic functions like in [10], because it contains non-computable functions. However, it is for us an open question whether a statement like (ChT) could be satisfied in a model of Coq, for instance in an effective model.

In such a case, the statement (ChT) would be independent of (axiom free) Coq: (ChT) would be both unprovable and unrefutable in Coq. We think (ChT) very much expresses an internal form of *Church thesis* in Coq: the functions which are typable in Coq are exactly the total recursive functions. The problem which such a statement is that the notion of totality is not independent from the logical framework in which such a totality is expressed and some frameworks are more expressive that others, e.g., Set theory defines more total recursive functions that Peano arithmetic. It is not clear how (ChT) could be used to simplify undecidability proofs in Coq.

## 8 Reifying Undecidable Predicates

In Sect. 3, we did explain how to reify the non-informative predicate $(\exists n, P\ n)$ into the informative predicate $\{n \mid P\ n\}$, for $P$ of type $\mathcal{N}\to$ `Prop`. This occurred under an important restriction: $P$ is assumed Coq-decidable there. The Coq term `nat_reify` that implements this transformation is nevertheless used in Sect. 6 to reify the undecidable "computes into" predicate `ra_bs`. This predicate is first represented as an existential quantification of the decidable precidate `ra_ca`, which is basically a bounded version of `ra_bs`. Then `nat_reify` is used to compute the bound by minimization. Without entering in the full details, we introduce some of the developments that can be found in the file applications.v.

We describe how to reify other kinds of undecidable predicates. For instance, we can reify undecidable predicates that can be bounded in some broad sense. Consider a predicate $P:X\to$ `Prop` for which we assume the following: $P$ is equivalent to $\bigcup_n (Q\ n)$ for some $Q:\mathcal{N}\to X\to$`Prop` such that $Q\ n$ is (informatively) finite for any $n:\mathcal{N}$. Then, the predicate $\exists P$ can be reified into $\Sigma P$:

| | |
|---|---|
| `Variables` | $(X:$`Type`$)\ \ (P:X\to$`Prop`$)\ \ (Q:\mathcal{N}\to X\to$`Prop`$)$ |
| | $(H_P:\forall x,\ P\ x \iff \exists n,\ Q\ n\ x)$ |
| | $\big(H_Q:\forall n,\ \{l:$`list`$\ X \mid \forall x,$ `In`$\ x\ l \iff Q\ n\ x\}\big)$ |
| `Theorem` | `weighted_reif`$:(\exists x:X, P\ x)\to\{x:X\mid P\ x\}$ |

The idea of the proof is simply that the first parameter of $Q$ is a weight of type $\mathcal{N}$ and that for a given weight $n$, there are only finitely many elements $x$ that satisfy $Q\,n\,x$ (hence $P\,x$). The weight $n$ such that $\exists x,\ Q\,n\,x$ is reified using

`nat_reify`, then the value $x$ is computed as the first element of the list given by $H_Q\ n$. The hypothesis $\exists x, P\,x$ ensures that the list given by $H_Q\ n$ is not empty.

Among its direct applications, such a weighted reification scheme can be used to reify *provability* predicates for arbitrary logics, at least those where formulæ and proofs can be encoded as natural numbers. This very low restriction allows to cover a very wide range of logics, with the notable exception of infinitary logics (where either formulæ are infinite or some rules have an infinite number of premisses). Hence, one can compute a proof of a statement provided such a proof exists. Another application is the reification of the *normalizable* predicate for any reduction (i.e. binary) relation which is finitary (i.e. with finite direct images). This applies in particular to $\beta$-reduction in $\lambda$-calculus.

To conclude, we implement a judicious remark of one of the reviewers. He points out that we can derive a proof of *Markov's principle for recursively enumerable predicates* over $\mathcal{N}^k$ (instead of just decidable ones). These are predicates of the form $\boldsymbol{v} \mapsto [\![f]\!]\ \boldsymbol{v}\ 0$ for some $\mu$-recursive $f$ function of arity k.

$$\texttt{Theorem re\_reify}\ k\ (f : \mathcal{A}_k) : \quad \left(\exists \boldsymbol{v} : \mathcal{N}^k,\ [\![f]\!]\ \boldsymbol{v}\ 0\right) \to \left\{\boldsymbol{v} : \mathcal{N}^k \mid [\![f]\!]\ \boldsymbol{v}\ 0\right\}$$

Hence if a recursively enumerable predicate can be proved inhabited, possibly using 1-consistent axioms in sort `Prop` such as e.g. excluded middle, then a witness of that inhabitation can be computed.

## 9 The Structure of the Coq Source Code

The implementation involves around 4 500 lines of Coq code. It has been tested and should compile under Coq 8.5pl3 and Coq 8.6. It is available under a Free Software license at https://github.com/DmxLarchey/Coq-is-total.

More than half of the code belongs to the utils.v utilities library, mostly in files pos.v, vec.v and tree.v. These could be shrunk further because they contain some code which is not necessary to fulfil the central goal of the paper. The files directly relevant to this development are:

**utils.v** The library of utilities that regroups notations.v, tac_utils.v, list_utils.v, pos.v, nat_utils.v, vec.v, finite.v and tree.v;

**nat_minimizer.v** The reification of $\exists P$ to $\Sigma P$ by unbounded minimization of decidable predicates of types $\mathcal{N} \to$ `Prop` and $\mathcal{N} \to$ `Type`, see Sect. 3;

**recalg.v** The dependently typed definition of recursive algorithms with a general recursion principle and the injectivity of type constructors, see Sect. 4;

**a_{rel,bs,ca}.v** The definitions of relational, big-step and cost aware big-step semantics, with inversion lemmas, see Sects. 5.1 and 5.2;

**ra_sem_eq.v** The proof of equivalence between the three previous semantics, see Sects. 5.1 and 5.2;

**ra_ca_props.v** High-level results about cost aware big-step semantics, mainly its functionality and its decidability, see Sect. 5.3;

**decidable_t.v** The decidability library to lift decision arguments to finitely quantified statements, see Sect. 5.3;

**coq_is_total.v** The file that implements Sect. 6, which shows that any provably total recursive function can be represented by a Coq term;

**applications.v** The file that implements Sect. 8, reification of (undecidable) weighted predicates, provability predicates, normalizability predicates and recursively enumerable predicates.

## References


1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004)
2. Bove, A., Capretta, V.: Modelling general recursion in type theory. Math. Struct. Comput. Sci. **15**(4), 671–708 (2005)
3. Castéran, P.: Utilisation en Coq de l'opérateur de description (2007). http://jfla.inria.fr/2007/actes/PDF/03_casteran.pdf
4. Coen, C.S., Valentini, S.: General recursion and formal topology. In: Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, EPiC Series, Edinburgh, UK, 15 July 2010, vol. 5, pp. 71–82. EasyChair (2010)
5. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press, New York (1989)
6. Krivine, J.: Lambda-Calculus, Types and Models. Ellis Horwood Series in Computers and Their Applications. Ellis Horwood, Masson (1993)
7. Larchey-Wendling, D.: A constructive mechanization of Lambda Calculus in Coq (2017). http://www.loria.fr/~larchey/Lambda_Calculus
8. Norrish, M.: Mechanised computability theory. In: Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 297–311. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22863-6_22
9. Soare, R.I.: Recursively Enumerable Sets and Degrees. Springer-Verlag New York Inc., New York (1987)
10. Werner, B.: Sets in types, types in sets. In: Abadi, M., Ito, T. (eds.) TACS 1997. LNCS, vol. 1281, pp. 530–546. Springer, Heidelberg (1997). doi:10.1007/BFb0014566
11. Xu, J., Zhang, X., Urban, C.: Mechanising turing machines and computability theory in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 147–162. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39634-2_13
12. Zammit, V.: A mechanisation of computability theory in HOL. In: Goos, G., Hartmanis, J., Leeuwen, J., Wright, J., Grundy, J., Harrison, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 431–446. Springer, Heidelberg (1996). doi:10.1007/BFb0105420
13. Zammit, V.: A proof of the S-m-n theorem in Coq. Technical report, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK, March 1997. http://kar.kent.ac.uk/21524/