Science of Computer Programming 126 (2016) 31–51

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Towards Turing computability via coinduction

Alberto Ciaffaglione

Dipartimento di Matematica e Informatica, Università di Udine, Italia

ARTICLE INFO

Article history:
Received 26 April 2015
Received in revised form 15 February 2016
Accepted 17 February 2016
Available online 27 February 2016

Keywords:
Program certification
Coq proof assistant
Coinductive types

ABSTRACT

We adopt corecursion and coinduction to formalize Turing Machines and their operational semantics in the Coq proof assistant. By combining the formal analysis of converging and diverging computations, via big-step and small-step predicates, our approach allows us to certify the correctness of concrete Turing Machines. An immediate application of our methodology is the proof of the undecidability of the halting problem, therefore our effort may be seen as a first step towards the formal development of basic computability theory.

1. Introduction

In this paper we present and discuss an encoding of *Turing Machines* (TMs) and their semantics in the Coq implementation of the *Calculus of (Co)Inductive Constructions* ($CC^{(Co)Ind}$). Actually, we do not find in the literature much formalization work dealing with *computability theory*, a foundational, major area of computer science, whereas several other domains have benefited, in recent years, from formal developments carried out within mechanized environments.

As far as we know, the most recent contributions are [11,3,1,13]. Norrish [11] develops a proof of equivalence between the recursive functions and the λ-calculus computational models, and formalizes some computability theory results in the HOL4 system. The two works most related to the present one are those focusing on TMs. Asperti and Ricciotti [1] develop computability theory up to the construction of a universal machine, by carrying out their effort from a perspective oriented to complexity theory in Matita. Xu, Zhang and Urban [13] prove the undecidability of the halting problem and relate TMs to register machines and recursive functions through a universal TM in Isabelle/HOL.

Actually, TMs form an object system which is challenging in several respects. If we consider just the objects and concepts involved in their semantics, we observe that the tape, used as workspace for computing, is *infinite* in both directions; moreover, running TMs may give rise to *diverging* computations. Hence, TMs provide with a scenario where the user is required to define and reason about *infinite* objects and concepts. To address formally such an object system, in this paper we settle within *Intuitionistic* (*a.k.a. Constructive*) *Type Theory*. This framework makes available *coinductive types*, *i.e.*, types that have been conceived to provide finite representations of infinite structures. In particular, a handy technique for dealing with *corecursive* definitions and *coinductive* proofs in $CC^{(Co)Ind}$ was introduced by Coquand [6] and refined by Giménez [7]. Such an approach is particularly appealing, because proofs carried out by coinduction are accommodated as any other infinite, corecursively defined object. This technique is mechanized in the Coq proof assistant [14].

Due to this basic choice, the present work is in fact a departure from the two cited formalizations of TMs: on the one hand, we adopt corecursion as a *definition* principle to deal with the infinity of tapes; on the other hand, we exploit coinduction as a *proof* principle to cope with the infinity of computations.

E-mail address: alberto.ciaffaglione@uniud.it.

http://dx.doi.org/10.1016/j.scico.2016.02.004
0167-6423/

In [11], Norrish advocates the formalization of the most operational models of computation, *i.e.*, TMs and register machines. However, a major difficulty arises in pursuing the goal, because these models (TMs, in the present case) have a low-level nature, and therefore are completely *non-structured*.

Apart from the `Coq` support for coinductive types, we see other reasons to carry out a formalization of TMs. As just observed, there are connections between TMs and low-level languages, such as the assembly and machine code. Moreover, as it is well-known, traditional papers and textbooks approach TMs at a more superficial level of detail, and in particular the arguments why individual TMs are correct are often left out. Therefore, the mechanization effort in a proof assistant, besides offering the possibility to discover errors (as happens in [13] for the textbook [2]), may typically improve our confidence in the details.

Inspired by our previous effort on unlimited register machines [3], we encode TMs and their operational semantics from the perspective of *program certification*: *i.e.*, we introduce and justify a methodology to prove the correctness of concrete TMs; then, we exploit such a machinery to formalize the *undecidability of the halting problem*. We remark that these kinds of proofs for concrete TMs are acknowledged in [13] as the most important contribution of that paper (where the undecidability of the halting problem is proved in `Isabelle/HOL`).

Besides being intellectually stimulating, our work offers us the possibility of popularizing corecursion and coinduction (because TMs are broadly known), an aim that we pursue by justifying the encoding methodology in an analytical way and via suggestive examples, that deal with a sort of "animation" of TMs.

We have used, as starting point for our development, the textbook by Cutland [8]. As an effort towards a broader audience, we display not only `Coq` code in this paper, but we present the encoding also at a more abstract level. The formalization is available as a web appendix [5] of the manuscript. The first part of the paper (till the end of Section 7) contains material from [4].

Synopsis In the next section we recap TMs, then in the two following sections we introduce their formalization and illustrate the implementation of coinduction in `Coq`. In the two central sections 5 and 6 we define a big-step operational semantics for TMs and address its adequacy via a small-step semantics, respectively. In the next "application" sections we prove the correctness of three representative TMs (Section 7), introduce the sequential composition of TMs (Section 8), and prove the undecidability of the halting problem (Section 9). In the last section we state final remarks and discuss related and future work.

2. Turing Machines

Turing Machines (TMs) [12], one among the frameworks proposed to set up a formal characterization of the intuitive ideas of computability and decidability, perform algorithms as carried out by a human agent using paper and pencil. In this work we address *deterministic*, single tape TMs, as defined by Cutland [8].

Alphabet and tape TMs operate on a *paper tape*, which is *infinite* in both directions and is divided into single squares along its length. Each square is either blank or contains a symbol from a *finite* set of symbols $s_0, s_1, \ldots, s_n$, named the *alphabet* $\mathcal{A}$ (in fact, the "blank" B is counted as the first symbol s_0).

Specification and computation At any time, TMs both scan a single square of the tape (via a *reading/writing head*) and are in one of a *finite* number of *states* $q_1, \ldots, q_m$. Depending on the current state q_i and the symbol being scanned s_h, TMs take *actions*, as indicated by a *specification*, *i.e.* a *finite* collection of quadruples $\langle q_i, s_h, q_j, x\rangle$, where $i, j \in [1..m]$, $h \in [0..n]$, and $x \in \{R, L\} \cup \mathcal{A}$, $R, L \notin \mathcal{A}$:

$$\langle q_i, s_h, q_j, x\rangle \triangleq \begin{array}{ll} 1) & \text{change the state from } q_i \text{ into } q_j \\ 2) & \textit{if } x{=}R \textit{ then} \text{ move the head one square to the right} \\ & \textit{else if } x{=}L \textit{ then} \text{ move the head one square to the left} \\ & \textit{else if } x{=}s_k\ (k{\in}[0..n]) \textit{ then} \text{ replace } s_h \text{ with } s_k \end{array}$$

As declared above, we deal with deterministic TMs, *i.e.*, the specifications are *non-ambiguous*: for every pair q_i, s_h there is at most one quadruple of the form $\langle q_i, s_h, q_j, x\rangle$. When provided with a tape, a specification becomes an *individual* TM, which is capable of performing a *computation*: namely, it keeps carrying out actions by starting from the initial state q_1 and the symbol scanned by the initial position of the head. Such a computation is said to *converge* if and only if, at some given time, there is no action specified for the current state q_i and the current symbol s_h (that is, there is no quadruple telling what to do next). On the other hand, if this never happens, such a computation is said to *diverge*.

Computable functions TMs may be regarded as devices for computing numerical *functions*, according to the following conventions.

A natural number m, seen as the *input* of a unary function, is represented on a tape by an amount of $m+1$ *consecutive* occurrences of the "tally" symbol 1 (with the head scanning the leftmost one) and the blank symbol elsewhere:

$$\cdots \mid \underbrace{\overset{\Downarrow}{1} \mid - \mid 1}_{m+1} \mid \cdots$$

According to this assumption, on the one hand the representation of the $0 \in \mathbb{N}$ is distinguished from the blank tape; moreover, both the initial position of the head and the initial tape are fixed, hence functions can be introduced.

Then, a machine M computes the *partial* function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ when, for every $a, b \in \mathbb{N}$, the computation under M, starting from its initial state and the leftmost 1 of the a representation, stops with a tape that contains a *total* of b symbols 1 (not necessarily consecutive) *if and only if* $a \in dom(f)$ and $f(a) = b$ (therefore, f is undefined on all inputs a that make the computation diverge). The n-ary partial functions $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ are computed in a similar way, where the representations of the n inputs are separated by single blank squares.

Consequently, computability theory can be developed via TMs, leading to the well-known characterization of the class of effectively computable functions.

3. Turing Machines in Coq

As described in the previous section, TMs are formed by two components: the specification and the tape, whose content in fact instantiates the former, making it executable. Specifications and tapes actually work together, but are independent of each other from the point of view of the formalization.

Our encoding of TMs in Coq reflects such an independence: in the present work we are mainly interested in the formal treatment of the tape, which is more problematic and particularly delicate; conversely, we do not pursue the specification-component management (*automata* are also supported by Coq's library), thus keeping that part of the formalization down to a minimum.

We remark that we will adopt two styles of presentation throughout the paper: genuine Coq code (when significant) and a more abstract notation, to skip unnecessary (because either obvious or cumbersome) technical details.

Specification and tape Concerning the *specification* part, we represent states via natural numbers, while alphabet symbols and operations performed by the head are finite collections of elements. We work with an alphabet of three symbols: the "blank" B and the "tally" 1, adopted in [8] (see the previous section), and a "mark" symbol 0, introduced as well in [8] to develop examples.[1] Finally, specifications are finite sequences (*i.e.*, lists) of actions (*i.e.*, quadruples)[2]:

State	:	p, q, i	$\in$	$\mathbb{N}=\{0, 1, 2, \ldots\}$	state
Sym	:	a, b	$\in$	$\{B, 1, 0\}$	alphabet symbol
Head	:	x, y	$\in$	$\{R, L, W(a)\}$	head operation
Act	:	α	$\in$	$State \times Sym \times State \times Head$	action
Spec	:	T, U, V	$::=$	$(\iota \mapsto \alpha_\iota)^{\iota \in [1..n]}$ $(n \in \mathbb{N})$	specification

It is worth mentioning that we formalize in Coq the above datatypes by means of *inductive types* (built-in natural numbers for *State* and lists for *Spec*), because Coq, being type theory-based, does not offer an efficient support for the treatment of *sets*. Actually, the encoding through inductive types gives us the possibility to argue by case analysis (and by induction) about their elements[3]:

```
Inductive nat: Set := 0: nat | S: nat -> nat. Definition State := nat.
Inductive Sym: Set := B: Sym | one: Sym | zero: Sym.
Inductive Head: Set := R: Head | L : Head | W: Sym -> Head.
Definition Spec: Set := (list (State * Sym * State * Head)).
```

To represent the *tape*, whose squares are scanned by the head and contain the alphabet symbols, we adopt a pair of *streams* (*a.k.a.* infinite sequences), a datatype borrowed from the Haskell community, where it is named "zipper":

HTape	:	l, r	$::=$	$(\iota \mapsto a_\iota)^{\iota \in [0..\infty]}$	half tape (stream)
Tape	:	s, t, u	$::=$	$\langle\!\langle\, l, r \,\rangle\!\rangle$	full tape (zipper)

[1] We stress that the formalization's metatheoretical properties investigated till the end of Section 6 are in fact independent from the particular choice of the alphabet, which could be equivalently taken as a parameter (see the discussion at the outset of Section 7).

[2] The middle columns display the metavariables and the datatypes they range over.

[3] Notice that we write one and zero in Coq to avoid collision with built-in natural numbers.

The intended meaning of this encoding is that the second stream ($r = r_0{:}r_1{:}\ldots$) models the infinity of the tape towards the right, while the first stream ($l = l_0{:}l_1{:}\ldots$) is infinite towards the left. At any time, the head "⇓" will be scrutinizing the first symbol of r, which corresponds physically to:

$$\cdots \mid l_1 \mid l_0 \mid \underset{\Downarrow}{r_0} \mid r_1 \mid \cdots$$

Streams of symbols are rendered in `Coq` via the following coinductive type[4]:

```
CoInductive HTape: Set := Cons: Sym -> HTape -> HTape.
```

It is apparent that this representation of the tape allows for a direct access to its content, an operation which has therefore minimal complexity (the first symbol of a stream is immediately available, see the next section for details).

Transitions To make specifications induce computations it is necessary, given the current state and tape symbol, to extract from such list-structures the corresponding target state and head operation. It may be the case that no action is specified for a given state-symbol pair (because the computation is intended to terminate there), therefore we need to introduce a *partially* defined *transition* function *tr*: $\mathit{Spec} \to \mathit{State} \to \mathit{Sym} \rightharpoonup (\mathit{State} * \mathit{Head})$:

$$\begin{aligned} tr(T, p, a) \;\triangleq\; & \texttt{match}\ T\ \texttt{with}\ [\,] \Rightarrow \uparrow \mid \langle q, b, i, y\rangle : U \Rightarrow \\ & \texttt{if}\ (q{=}p \wedge b{=}a)\ \texttt{then}\ (i, y)\ \texttt{else}\ tr(U, p, a) \end{aligned}$$

Notice that "[ ]" and ":" represent the `nil` and `cons` constructors for lists, and `match` is the standard destructor for (co)inductive types (again, see the next section). A specification T is scanned by the recursive function *tr* from left to right: given an input pair (p, a), the target state and head operation are obtained from the *first* (*i.e.*, leftmost) quadruple of shape $\langle p, a, q, x\rangle$ found in T (no matter if there are others in the form $\langle p, a, i, y\rangle$); when, conversely, there is no corresponding quadruple in T, *tr* is undefined, written "↑". In this way, we delegate to the *tr* function the management of the TMs' *determinism*; in other words, we get rid of introducing an extra predicate to check the "well-formedness" of specifications: simply, we examine them from left to right.

We formalize *tr* as a *total* function (functions *must* be total in `Coq`) by means of the `option` type, which returns the special value `None` for "↑" and `(Some _)` for genuine values. As displayed above, *tr* is a recursive function which consumes objects belonging to *Spec*, therefore we encode it via the `Fixpoint` command:

```
Fixpoint tr (T:Spec) (p:State) (a:Sym): option (State * Head):= ...
```

As declared at the beginning of the section, the specification component of TMs is not our main concern, hence we keep its formalization as minimal as possible (via lists and related machinery). Our choice may be seen as an evidence of the independence of the tape and the specification from each other; in fact, we will not investigate the metatheory of the latter before Section 8.

4. Coinduction in Coq

The `Coq` proof assistant [14] supports the formal treatment of circular, infinite data and relations by means of the mechanism of *coinductive types*. In this section we provide an introduction to the topic, to convey to the reader the fundamental insights about the formalization presented in this paper.

Infinite objects One may formalize concrete, infinite *objects* (*i.e.*, data) as elements of coinductive *sets*,[5] which are fully described by a set of *constructors*. From a pure logical point of view, the constructors can be seen as *introduction rules*; these are interpreted coinductively, that is, they are applied infinitely many times, hence the type being defined is inhabited by infinite objects. We take as working example the coinductive set `HTape`, that we have introduced in the previous section via the `Cons` constructor and the alphabet `Sym`:

```
CoInductive HTape: Set := Cons: Sym -> HTape -> HTape.
```

Once a new coinductive type is defined, the system supplies automatically the *destructors*, *i.e.*, an extension of the native pattern-matching capability, to *consume* the elements of the type itself. Therefore, coinductive types can also be viewed as the *largest* collection of objects closed *w.r.t.* the destructors. We use here the standard `match` destructor to extract the *head* and *tail* from streams:

[4] While inductive types are inhabited by *finite*-size elements, coinductive types give rise to *infinite* objects (see the next section, which is devoted to coinduction in `Coq`).

[5] Coinductive sets are coinductive types whose type is the sort `Set`.

```
Definition hd (h:HTape) := match h with | Cons a k => a end.
Definition tl (h:HTape) := match h with | Cons a k => k end.
```

However, the destructors *cannot* be used for defining functions by *recursion* on coinductive types, because their elements cannot be consumed down to a base case. The natural way to allow self-reference with coinductive types is in fact the *dual* approach of *building* objects that belong to them. Such a goal is fulfilled by defining *corecursive* functions, like, *e.g.*, the following ones:

```
CoFixpoint Bs := Cons B Bs.
CoFixpoint same (a:Sym) := Cons a (same a).
CoFixpoint blink (a b:Sym) := Cons a (Cons b (blink a b)).
CoFixpoint merge (h k:HTape) := match h with | Cons a h' =>
  match k with | Cons b k' => Cons a (Cons b (merge h' k')) end end.
```

Corecursive functions are introduced via the `CoFixpoint` command and look as recursive definitions; they yield infinite objects that belong to a coinductive type and may have any type as domain (notice that in the last definition the two parameters are infinite objects as well). Above, we have declared corecursive functions that build streams as follows: `Bs` is the infinite repetition of the blank symbol; `same` is more general, as it takes its parameter from input; `blink` is even more general, using its two input parameters in alternation; `merge` destructs two infinite parameters and puts their elements together, in alternation.

To be accepted by `Coq`, corecursive functions must satisfy a *guard condition*: *i.e.*, every corecursive call has to be guarded by *at least* one constructor[6] (`Cons` in the definitions above) and by *nothing but* constructors. In practice, corecursive functions are never unfolded, unless their elements are explicitly needed, "on demand", by a destruction operation (performed by the `match` command). This way of regulating the implementation of corecursion is inspired by *lazy* functional languages, where the constructors do not evaluate their arguments.

Infinite concepts Given a coinductive set (such as `HTape`), no *proof principle* can be automatically generated by the `Coq` system: actually, proving properties about infinite objects requires the potential of building *proofs* which are infinite too. What is needed is the design of *ad-hoc* coinductive *predicates*[7] (*i.e.*, relations), which are in fact types inhabited by corecursive *proof terms*. The traditional example is *bisimilarity*, that we define here on streams in `HTape`:

```
CoInductive EqH: HTape -> HTape -> Prop :=
             eqh: forall a:Sym, forall h k:HTape,
             EqH h k -> EqH (Cons a h) (Cons a k).
```

We introduce also an extra, more abstract notation for defining predicates (bisimilarity in the case, that we name $\simeq \subseteq$ *HTape* $\times$ *HTape*), with the goal of providing an alternative to `Coq` code, to be used in the rest of the paper:

$$\frac{a \in Sym \quad h,k \in HTape \quad h \simeq k}{a{:}h \simeq a{:}k}\ (\simeq)_\infty$$

The relation `EqH` (*i.e.*, $\simeq$ in the alternative notation, where the `Cons` constructor is represented by the overloaded symbol ":") amounts to the conclusions of infinite derivation trees built from the `eqh` rule ($(\simeq)_\infty$, respectively).

Actually, two streams are bisimilar if we can observe that their heads coincide and, recursively, *i.e.*, *coinductively*, their tails are bisimilar. Once this new predicate is defined, the system provides a corresponding proof principle to carry out proofs about bisimilarity: such a tool, named "guarded induction" principle [6,7], is particularly appealing in a context where proofs are managed as any other infinite object. In fact, a bisimilarity proof is just an infinite term built by corecursion via the `eqh` constructor (hence, it must respect the same guard constraint as have corecursive functions).

The guarded induction principle supplies a handy technique for building proofs inhabiting coinductive predicates, as such proofs can be carried out *interactively* through the `cofix` tactic.[8] This tactic allows the user to yield *infinitely regressive* proofs, by assuming the thesis as an extra hypothesis and using it later with care, *i.e.*, provided its application is guarded by constructors. In this way the user is not required to pick out a bisimulation beforehand (as in alternative approaches to coinduction), but may build it incrementally, via tactics.

To illustrate the support provided by the `cofix` tactic to argue by coinduction, we detail here the proof of $\forall a, b{\in}Sym.\ merge(same(a), same(b)) \simeq blink(a, b)$. First, the goal itself, playing the role of the coinductive hypothesis, is assumed among the hypotheses via the `cofix co_hp` command:

[6] Syntactically, the constructors guard the corecursive call "on the left"; this captures the intuition that infinite objects are built via the repetition of a "productive" step.

[7] Coinductive predicates are coinductive types whose type is the sort `Prop`.

[8] A tactic is a command to solve a goal or decompose it into simpler goals.

```
co_hp: forall a b: Sym, EqH (merge (same a) (same b)) (blink a b)
=========================================
forall a b:Sym, EqH (merge (same a) (same b)) (blink a b)
```

Then we introduce in the derivation context the two metavariables a, b (`intro` tactic) and we unfold the corecursive functions `same`, `merge` and `blink`, in turn, to perform a computation step, via suitable "rewriting" technical lemmas:

```
co_hp: forall a b: Sym, EqH (merge (same a) (same b)) (blink a b)
a : Sym
b : Sym
=========================================
EqH (Cons a (Cons b (merge (same a) (same b)))) (Cons a (Cons b (blink a b)))
```

Finally, the `eqh` constructor may be applied twice (`apply eqh` command), thus reducing the initial goal to an instance of the coinductive hypothesis:

```
co_hp: forall a b: Sym, EqH (merge (same a) (same b)) (blink a b)
a : Sym
b : Sym
=========================================
EqH (merge (same a) (same b)) (blink a b)
```

Now, we are eventually allowed to exploit (via the `apply` tactic) such a hypothesis `co_hp`, whose application is in fact guarded by the `eqh` constructor: this property follows from the fact that between the assumption of the coinductive hypothesis and its application we have only used `eqh` and unfolded corecursive functions, which is obviously admitted. The application of the coinductive hypothesis actually accomplishes the goal: intuitively, it has the effect of repeating *ad infinitum* the initial fragment of the proof, thus realizing the "and so on forever" motto. From a technical point of view, the application of `eqh` and the subsequent call to the `co_hp` coinductive hypothesis correspond to the construction, via corecursion, of the proof term inhabiting the property to be proved, where the `co_hp` application is therefore a guarded corecursive call.

We display now the whole proof, developed above via interactive tactics, in a *natural deduction* style,[9] with the aim of representing it in an equivalent but more compact way, to be exploited fruitfully in the rest of the paper:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\forall a, b{\in}Sym.\ merge(same(a), same(b)) \simeq blink(a,b)]_{(1)}}{\vdots}}{merge(same(a), same(b)) \simeq blink(a,b)}}{a{:}b{:}merge(same(a), same(b)) \simeq a{:}b{:}blink(a,b)}\ (\simeq)_\infty}{merge(a{:}same(a), b{:}same(b)) \simeq a{:}b{:}blink(a,b)}\ \text{(def: merge)}}{merge(same(a), same(b)) \simeq blink(a,b)}\ \text{(def: same, blink)}}{\forall a, b{\in}Sym.\ merge(same(a), same(b)) \simeq blink(a,b)}\ a, b{\in}Sym\ (1), \text{(introduction)}$$

It is apparent that `Coq` allows just a limited form of coinduction, which is disciplined by means of a *syntactic* check. We refer the reader to [9], where an alternative, *semantic* approach to guardedness has been recently proposed.

5. Operational semantics

As stressed in Sections 2 and 3, the semantics of TMs' specifications is parametric *w.r.t.* tapes: computations, induced by specifications, may either converge or diverge, depending on the tape coupled to them and the initial position of the head (once fixed the initial state). In Section 3 we have also chosen an encoding for tapes (via a zipper, made of two streams) such that the position of the head is implicit within the tape itself. Therefore, the semantics of TMs may be defined by considering configuration-triples (T, p, s), where T is a specification, p a state, and $s{=}\langle\!\langle l, r{=}r_0{:}r_1{:}\ldots\rangle\!\rangle$ a tape. Some configurations make actually a computation stop, because there is no action specified by T for the current state p and symbol r_0: these ones will play the role of the *values* of our semantics. In the following, with abuse of notation, we will represent by $tr(T, p, s)$ the application of the transition function tr, introduced in Section 3: in detail, we write $tr(T, p, s)\downarrow$ for stopping and $tr(T, p, s){\mapsto}\langle i, x\rangle$ otherwise.

In this section we define a *big-step* semantics for TMs, which will play the role of our main tool throughout the rest of the paper. The *potential* divergence of computations provides us with a typical scenario which may benefit from

[9] As usual, local hypotheses are indexed with the rules they are discharged by. Notice in particular that, according to Gentzen's notation, we write the coinductive hypothesis (among the leaves of the proof tree) within square brackets, to bear in mind that it is *discharged*, *i.e.*, canceled, in the course of a formal proof, because it represents a *local* hypothesis.

the use of *coinductive* specification and proof principles. In fact, a faithful encoding has to reflect the separation between converging and diverging computations, through two different judgments. Hence, we define the *inductive* predicate $b_* \subseteq Spec \times Tape \times State \times Tape \times State$ to cope with converging evaluations, and the *coinductive* $b_\infty \subseteq Spec \times Tape \times State$ to deal with diverging ones. Actually, we write more suggestively the instances of the two predicates as follows: $(s, p) \Rightarrow_T^* (t, q)$ stands for $b_*(T, s, p, t, q)$ and $(s, p) \Rightarrow_T^\infty$ for $b_\infty(T, s, p)$.

Definition 1 *(Evaluation).* Assume $T \in Spec$, $s = \langle\!\langle l = l_0 : l_1 : \ldots, r = r_0 : r_1 : \ldots \rangle\!\rangle$ and $t \in Tape$, $p, q, i \in State$. Then, b_* is defined by the following inductive rules:

$$\frac{tr(T, p, s) \downarrow}{(s, p) \Rightarrow_T^* (s, p)}\ (\text{stop}) \qquad \frac{tr(T, p, s) \mapsto \langle i, R \rangle \quad (\langle\!\langle r_0 : l, tl(r) \rangle\!\rangle, i) \Rightarrow_T^* (t, q)}{(\langle\!\langle l, r \rangle\!\rangle, p) \Rightarrow_T^* (t, q)}\ (\text{right})_*$$

$$\frac{tr(T, p, s) \mapsto \langle i, L \rangle \quad (\langle\!\langle tl(l), l_0 : r \rangle\!\rangle, i) \Rightarrow_T^* (t, q)}{(\langle\!\langle l, r \rangle\!\rangle, p) \Rightarrow_T^* (t, q)}\ (\text{left})_*$$

$$\frac{tr(T, p, s) \mapsto \langle i, W(a) \rangle \quad (\langle\!\langle l, a : tl(r) \rangle\!\rangle, i) \Rightarrow_T^* (t, q)}{(\langle\!\langle l, r \rangle\!\rangle, p) \Rightarrow_T^* (t, q)}\ (\text{write})_*$$

And b_∞ is defined by the following rules, (this time) interpreted coinductively[10]:

$$\frac{tr(T, p, s) \mapsto \langle q, R \rangle \quad (\langle\!\langle r_0 : l, tl(r) \rangle\!\rangle, q) \Rightarrow_T^\infty}{(\langle\!\langle l, r \rangle\!\rangle, p) \Rightarrow_T^\infty}\ (\text{right})_\infty$$

$$\frac{tr(T, p, s) \mapsto \langle q, L \rangle \quad (\langle\!\langle tl(l), l_0 : r \rangle\!\rangle, q) \Rightarrow_T^\infty}{(\langle\!\langle l, r \rangle\!\rangle, p) \Rightarrow_T^\infty}\ (\text{left})_\infty$$

$$\frac{tr(T, p, s) \mapsto \langle q, W(a) \rangle \quad (\langle\!\langle l, a : tl(r) \rangle\!\rangle, q) \Rightarrow_T^\infty}{(\langle\!\langle l, r \rangle\!\rangle, p) \Rightarrow_T^\infty}\ (\text{write})_\infty$$

Notice that in these rules we write r_0 and l_0 for $hd(r)$ and $hd(l)$, respectively (the functions *hd* and *tl* return the head and tail of streams, see Section 4). □

In our semantics, given a specification T, a tape s and a state p, we capture on the one hand the *progress* of both the head and the states' transitions, and on the other hand the *effect* of the operations performed by the head itself.

In detail, the intended meaning of $(s, p) \Rightarrow_T^* (t, q)$ is that the computation under the specification T, starting from the tape s and the state p, *stops* in the state q, transforming s into t. Conversely, $(s, p) \Rightarrow_T^\infty$ asserts that the computation under T, starting from s and p, *loops*: in this case a *final* tape cannot exist, because the initial s is scrutinized (and possibly updated) "ad infinitum".

Since TMs are not structured, we have embedded in the big-step semantics an alternative *structuring criterion*, *i.e.*, the height of the derivation tree. In fact, we have defined a base (*i.e.*, non-recursive) rule for b_* (the computation stops because no next action exists) and (co)inductive rules for both b_* and b_∞, to address how moving the head and writing on the tape is carried out within a converging computation and a diverging one, respectively.

We formalize the two predicates in `Coq` by means of the `Inductive` command for convergence and the `CoInductive` one for divergence (we do not show their body, as it is the straightforward translation of the rules displayed above):

```
Inductive bf: Spec -> Tape -> State -> Tape -> State -> Prop := ...
CoInductive bi: Spec -> Tape -> State -> Prop := ...
```

Discussion We complete this section with a digression about the choice of the two big-step predicates b_*, b_∞ and about their formal relationship.

Usually, big-step semantics describes only finite and successful evaluation of programs (it is the case of the b_* judgment, in our approach), failing to distinguish between diverging evaluations and those that "go wrong". Since TMs evaluations cannot go wrong (essentially because TMs are low-level), we have introduced the second judgment b_∞ to deal with all the extra evaluations, that coincide with diverging ones. This choice supports our goal of addressing the certification of the functions computed by TMs, which are in fact partially defined; that is, after inspecting concrete TMs and figuring out

[10] The relation b_∞ amounts to the conclusions of infinite derivation trees built from the above rules or, equivalently, it is the greatest fixed-point of such rules (see Section 4).

whether their evaluation converges or diverges, we are allowed to pick out the right predicate (either b_* or b_∞) and develop formal proofs about convergence or divergence.

We believe that our approach, besides being elegant, points out that the two predicates b_* and b_∞ "complement" each other; this insight may be made formal by establishing that they are mutually exclusive, *i.e.*, incompatible.

Proposition 1 *(Duality). Assume $T \in Spec$, $s, t \in Tape$, and $p, q \in State$.*

1. *If* $(s, p) \Rightarrow_T^* (t, q)$, *then* $\neg(s, p) \Rightarrow_T^\infty$
2. *If* $(s, p) \Rightarrow_T^\infty$, *then* $\neg(s, p) \Rightarrow_T^* (t, q)$

Proof. 1) By induction on the derivation of the first hypothesis and inversion of the second one (actually, $\neg(s, p) \Rightarrow_T^\infty$ can be managed as "if $(s, p) \Rightarrow_T^\infty$ then *False*"). 2) Immediate by point (1), again expanding the "¬" operator. □

We remark that Coq's proofs are very similar to those that one would develop working with paper and pencil, and are carried out by invoking a few tactics.

6. Adequacy

In this section we address two topics that strengthen the formalization developed so far. First, we give feedback about a different encoding for tapes that we pursued in a preliminary phase of our research. Moreover, we introduce a small-step semantics, which provides us with an additional tool for TMs.

Streams vs. lists Even if streams are a datatype which captures promptly and naturally the infinity of tapes, an encoding approach via (finite) *lists* may also be developed: in this case, the empty list is intended to represent an infinite sequence of blanks. The choice of lists makes explicit the assumption about TMs that, when a computation starts, only a finite number of squares can contain non-blank symbols (in fact, the representation of numerical functions in Cutland's setting, that we have adopted in Section 2, respects such a constraint).

Therefore, we proceed by encoding the tape through a pair of lists:

$$\begin{array}{lcll} HTape_L & : & ll, rl & ::= \quad (\iota \mapsto a_\iota)^{\iota \in [1..n]} \qquad \text{half tape (list, } n \in \mathbb{N}) \\ Tape_L & : & sl, tl & ::= \quad \langle\!\langle\, ll, rl \,\rangle\!\rangle \qquad\qquad\quad \text{full tape (list-pair)} \end{array}$$

Afterwards, big-step semantics predicates, playing the role of those dealing with streams in Section 5, can be introduced. However, since lists (conversely to streams) might be empty, such predicates must take into consideration this extra pattern and manage it via additional rules. The rules for R, *e.g.*, are as follows (where tr_L is the transition function version that works with list-tapes):

$$\frac{tr_L(T, p, \langle\!\langle\, ll, [\,] \,\rangle\!\rangle) \mapsto \langle i, R \rangle \quad (\langle\!\langle\, B{:}ll, [\,] \,\rangle\!\rangle, i) \Rightarrow_T^{L*} (t, q)}{(\langle\!\langle\, ll, [\,] \,\rangle\!\rangle, p) \Rightarrow_T^{L*} (t, q)}\ (r_{[\,]})_*$$

$$\frac{tr_L(T, p, \langle\!\langle\, ll, a{:}rl \,\rangle\!\rangle) \mapsto \langle i, R \rangle \quad (\langle\!\langle\, a{:}ll, rl \,\rangle\!\rangle, i) \Rightarrow_T^{L*} (t, q)}{(\langle\!\langle\, ll, a{:}rl \,\rangle\!\rangle, p) \Rightarrow_T^{L*} (t, q)}\ (r_L)_*$$

The inductive convergence predicate $b_{L*} \subseteq Spec \times Tape_L \times State \times Tape_L \times State$, whose instances, as usual, we write $(sl, p) \Rightarrow_T^{L*} (tl, q)$, has the same intended meaning of b_*. The coinductive divergence predicate $b_{L\infty} \subseteq Spec \times Tape_L \times State$, corresponding to b_∞ and written $(sl, p) \Rightarrow_T^{L\infty}$, is defined analogously.

By using the predicates b_{L*} and $b_{L\infty}$, we can prove that the semantics with streams may mimic that with lists, and a limited form of the opposite result (below, we denote with *Bs* the stream of blank symbols formalized in Section 4, and with "::" a recursive function that appends a list in front of a stream).

Proposition 2 *(Tape). Assume $T \in Spec$, $ll, rl, ll', rl' \in HTape_L$, and $p, q \in State$.*

1. *If* $(\langle\!\langle\, ll, rl \,\rangle\!\rangle, p) \Rightarrow_T^{L*} (\langle\!\langle\, ll', rl' \,\rangle\!\rangle, q)$, *then* $(\langle\!\langle\, ll{::}Bs, rl{::}Bs \,\rangle\!\rangle, p) \Rightarrow_T^* (\langle\!\langle\, ll'{::}Bs, rl'{::}Bs \,\rangle\!\rangle, q)$
2. $(\langle\!\langle\, ll, rl \,\rangle\!\rangle, p) \Rightarrow_T^{L\infty}$ *if and only if* $(\langle\!\langle\, ll{::}Bs, rl{::}Bs \,\rangle\!\rangle, p) \Rightarrow_T^\infty$

Proof. 1) By structural induction on the hypothetical derivation. 2) Both directions are proved by coinduction and hypothesis inversion. □

The impossibility of proving the reverse implication of point (1) depends on the fact that the tape encoding through lists is not unique, because one may append to any list blank symbols at will; hence, it would be necessary to introduce an

equivalence relation on list-tapes to develop their metatheory. For this reason (and since lists demand to double the length of proofs, as their predicates have two constructors for any action), we prefer working with streams.

Precisely, the technical benefit of stream-tapes is that the blank stream `Bs` can be freely *unfolded* via the straightforward lemma `Bs=(Cons B Bs)`, whereas, obviously, the empty list `nil` cannot be rewritten as `(cons B nil)`.

Small-step semantics We introduce now a *small-step* semantics *à la* Leroy [10], and prove that it is equivalent to the big-step one. The importance of formalizing a second semantics and proving such an equivalence is twofold: on the one hand, it raises our confidence in the soundness of the big-step semantics; moreover, it supplies us with an extra tool for addressing the certification of TMs.

First, we define a *one-step* reduction concept $\rightarrow$, to express the three basic actions of TMs (*i.e.*, moving the reading head and writing on the current square). Note (again) that, since TMs are not structured, we do not need to define *contextual* reduction rules. Then, we can formalize the small-step semantics as *sequences* of reductions: *finite* reductions $\overset{*}{\rightarrow}$, defined by *induction*, are the reflexive transitive closure of $\rightarrow$, while *infinite* reductions $\overset{\infty}{\rightarrow}$, defined by *coinduction*, are its transitive closure. The first two reductions are defined on $Spec \times Tape \times State \times Tape \times State$ and the third on $Spec \times Tape \times State$, but, as usual, we write their instances $(s, p) \rightarrow_T (t, q)$, $(s, p) \overset{*}{\rightarrow}_T (t, q)$, and $(s, p) \overset{\infty}{\rightarrow}_T$.

Definition 2 *(Reduction).* Assume $T \in Spec$, $s = \langle\!\langle l, r \rangle\!\rangle \in Tape$, and $p, q \in State$. Then, the *one-step* reduction $\rightarrow$ is defined by the following rules:

$$\frac{tr(T, p, s) \mapsto \langle q, R \rangle}{(\langle\!\langle l, r \rangle\!\rangle, p) \rightarrow_T (\langle\!\langle r_0{:}l, tl(r) \rangle\!\rangle, q)} \ (\rightarrow_R)$$

$$\frac{tr(T, p, s) \mapsto \langle q, L \rangle}{(\langle\!\langle l, r \rangle\!\rangle, p) \rightarrow_T (\langle\!\langle tl(l), l_0{:}r \rangle\!\rangle, q)} \ (\rightarrow_L)$$

$$\frac{tr(T, p, s) \mapsto \langle q, W(a) \rangle}{(\langle\!\langle l, r \rangle\!\rangle, p) \rightarrow_T (\langle\!\langle l, a{:}tl(r) \rangle\!\rangle, q)} \ (\rightarrow_W)$$

For $t, u \in Tape$, $i \in State$, *finite* reduction $\overset{*}{\rightarrow}$ is defined by induction, via the rules:

$$\frac{}{(s, p) \overset{*}{\rightarrow}_T (s, p)} \ (\overset{*}{\rightarrow}_0) \qquad \frac{(s, p) \rightarrow_T (u, i) \quad (u, i) \overset{*}{\rightarrow}_T (t, q)}{(s, p) \overset{*}{\rightarrow}_T (t, q)} \ (\overset{*}{\rightarrow}_+)$$

And *infinite* reduction $\overset{\infty}{\rightarrow}$ is defined by the following coinductive rule:

$$\frac{(s, p) \rightarrow_T (t, q) \quad (t, q) \overset{\infty}{\rightarrow}_T}{(s, p) \overset{\infty}{\rightarrow}_T} \ (\overset{\infty}{\rightarrow}_\infty)$$

It turns out that evaluation and reduction are equivalent concepts, both in their converging and diverging versions. We remark that our proofs are *constructive*, whereas Leroy [10] had to postulate the "excluded middle" for divergence. Notice that we state a preliminary lemma that collects all the properties involving the reduction that will be useful in the rest of the paper.

Lemma 1 *(Auxiliary). Assume $T \in Spec$, $s, t, u \in Tape$, and $p, q, i \in State$.*

1. *If $(s, p) \overset{*}{\rightarrow}_T (u, i)$ and $(u, i) \overset{*}{\rightarrow}_T (t, q)$, then $(s, p) \overset{*}{\rightarrow}_T (t, q)$*
2. *If $(s, p) \rightarrow_T (u, i)$ and $(u, i) \Rightarrow^*_T (t, q)$, then $(s, p) \Rightarrow^*_T (t, q)$*
3. *If $(s, p) \overset{*}{\rightarrow}_T (u, i)$ and $(u, i) \Rightarrow^*_T (t, q)$, then $(s, p) \Rightarrow^*_T (t, q)$*

Proof. 1) By induction on the structure of the derivation of the first hypothesis. 2) By inversion of the first hypothesis. 3) By induction on the structure of the derivation of the first hypothesis and point (2). □

Proposition 3 *(Equivalence). Assume $T \in Spec$, $s, t \in Tape$, and $p, q \in State$.*

1. *$(s, p) \Rightarrow^*_T (t, q)$ if and only if $(s, p) \overset{*}{\rightarrow}_T (t, q)$ and $tr(T, q, t)\downarrow$*
2. *$(s, p) \Rightarrow^\infty_T$ if and only if $(s, p) \overset{\infty}{\rightarrow}_T$*

Proof. 1) Both directions are proved by structural induction on the hypothetical derivation; the direction ($\Leftarrow$) requires also Lemma 1(2). 2) Both directions are proved by coinduction and hypothesis inversion. □

Again, Coq's proofs are very similar to "paper and pencil" ones, via a few tactics. The Proposition points out that the proof practice of reduction and evaluation is very similar in Coq; however, the small-step predicates are slightly less handy than their big-step counterparts. In fact, when performing a TM action in top-down fashion by means of Coq's proof editor, the user is required to exhibit the witness tape, besides the target state (compare, for convergence, the rule $(\xrightarrow{*}_{+})$ in Definition 2 with any of the rules $(right)_*$, $(left)_*$, $(right)_*$ in Definition 1, bearing in mind that all have to be used for reducing to premises starting from conclusions). Moreover, the small-step version lacks the "halting" concept (*i.e.*, $tr(T, q, t)\downarrow$), which is internalized by the big-step judgment.

7. Certification

In this section we use the big-step predicates b_* and b_∞, introduced in Section 5 and justified in Section 6, to address the *certification* of the partial functions computed by *individual* TMs. We remark that, to pursue such a goal, it is mandatory to *implement* in Coq not only the tape (that we have deeply investigated in previous sections), but also the transition function *tr* and therefore its related datatypes, *i.e. Sym* (the alphabet) and *State* (the states).

Actually, in Section 3 we have chosen an implementation for these datatypes, but we want to stress now that all the metatheory developed so far is *independent* from the particular choice of *tr*, *Sym*, *State*. In other words, we could assume them just as parameters (*i.e.*, to fix only their type) and this would lead to develop the same metatheory (we refer the reader to the web appendix [5]):

```
Parameter Sym: Set. Parameter State: Set.
Parameter tr: Spec -> State -> Sym -> option (State * Head).
```

Preliminaries TMs' divergence may be caused by different kinds of behavior. Clearly, it is easy to manage the scenario where a finite portion of the tape is scanned. The interesting case is when TMs scrutinize an infinite area of it; this may happen by moving the head infinitely either just in one direction or in both directions. In this section we address one example for each pattern of behavior, to convey to the reader the confidence that we can master all of them.

We remark that our definition of semantics does not force the states to play particular roles: in principle, any state may be used to start a computation; similarly, being our transition function *tr* partially defined (see Section 3), every state may potentially behave as a final one. This freedom can be seen as the peculiar characteristic of our approach: we try to keep the formalization as general as possible, tuning it only when demanded by a particular application.

Notice that we will address the examples in this section by using state 1 as the initial (and smallest) state for TMs, to adhere to Cutland's setting [8].

From now on, we will make use of the following notation and terminology:

- On a tape, "$a \mid -$" represents an infinite amount of "a" symbols towards the right, and "$- \mid a$" towards the left.
- When a specification T is fixed, a *configuration* is a pair $\langle state, tape\rangle$.

First example: R moves The first partial function that we work out computes the half of *even* natural numbers, and is not defined on *odd* ones:

$$div2(n) \triangleq \begin{cases} n/2 & \text{if } n \in \mathbb{E} \\ \uparrow & \text{if } n \in \mathbb{O} \end{cases}$$

We remember, from Section 2, that an input n is represented by the tape:

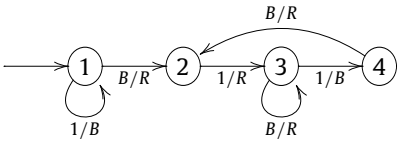
$$- \mid B \mid \underbrace{\overset{\Downarrow}{1} \mid - \mid 1}_{n+1} \mid B \mid - \tag{1}$$

which is formalized as $\langle\!\langle Bs, 1{:}ones(n){::}Bs\rangle\!\rangle$, where Bs is the blank stream, $ones(n)$ a list of n consecutive "1s", "::" the recursive function app_ls that appends a list in front of a stream, and ":" the Cons constructor of streams:

```
Fixpoint ones (n:nat): list Sym := match n with
         | 0 => nil | (S m) => (cons one (ones m)) end.
Fixpoint app_ls (l:list Sym) (s:HTape): HTape := match l with
         | nil => s | (cons b l') => (Cons b (app_ls l' s)) end.
```

One algorithm that implements the *div2* function is conceived as follows. Erase the first "1" (which occurs by definition) and move to the right; then try to find pairs of consecutive "1": if you succeed, erase the second "1" and restart the cycle, otherwise (a single "1" is found) move indefinitely to the right.

Such an algorithm can be realized, *e.g.*, by the following specification T:



where the edges are labeled with pairs "symbol/action" (but we write a for $W(a)$), and the roles of the states are as follows:

(1) starts the computation, by erasing the first "1" and moving to state 2;
(2) represents that an even number of "1s" has been read (and half of them erased), acting as the unique final state if n is even;
(3) either behaves as a looping state, moving the head forever to the right (n is odd), or it represents that the first "1" out of a consecutive pair of "1s" has been read (scanning the second "1" leads to erase it and move to state 4);
(4) represents that an even number of "1s" has been read and half of them erased, therefore it moves to state 2 to restart the cycle.

The implementation T of the *div*2 function may be certified using the predicates b_* and b_∞. To fulfill our goal we carry out in `Coq`, via tactics, a top-down formal development that simulates the computation of the TM at hand.

The evaluation starts from state 1 and the tape shown in (1); first, we perform a write-B and a move-R actions, thus reaching state 2 with the tape:

$$- \mid B \mid \underbrace{\overset{\Downarrow}{1} \mid - \mid 1}_{\text{n}} \mid B \mid - \tag{2}$$

Proving the *divergence* requires a combination of coinductive and inductive reasoning. The core property is divergence when proceeding from state 3 and a right-hand blank tape, a property which is proved by coinduction[11]:

$$\dfrac{l \in HTape \qquad \dfrac{tr(T,3,\langle\!\langle l, B{:}Bs\rangle\!\rangle) = \langle 3, R\rangle \qquad \dfrac{[\forall l \in HTape.\,(\langle\!\langle l, Bs\rangle\!\rangle, 3) \Rightarrow_T^\infty]_{(1)} \quad \vdots \quad (\langle\!\langle B{:}l, Bs\rangle\!\rangle, 3) \Rightarrow_T^\infty}{(\langle\!\langle l, B{:}Bs\rangle\!\rangle, 3) \Rightarrow_T^\infty}\,(\text{right})_\infty}{(\langle\!\langle l, Bs\rangle\!\rangle, 3) \Rightarrow_T^\infty}\,(\text{def: Bs})}{\forall l \in HTape.\,(\langle\!\langle l, Bs\rangle\!\rangle, 3) \Rightarrow_T^\infty}\,(1), (\text{introduction}) \tag{3}$$

If n in (2) is *odd*, we show that it leads to divergence (by induction on k):

$$\forall k \in \mathbb{N}, \forall l \in HTape.\,(\langle\!\langle\, l,\ ones(2k{+}1){::}Bs\,\rangle\!\rangle, 2) \Rightarrow_T^\infty$$

If $k = 0$, carry out a move-R action and use the proof (3); if $k = h{+}1$, complete a cycle (erasing the second "1") and conclude via the induction hypothesis.

We address the *convergence* in the complementary scenario (an *even* input n in (2)) by proving the following property, again by induction on k:

$$\forall k \in \mathbb{N}, \forall l \in HTape.\,(\langle\!\langle\, l,\ ones(2k){::}Bs\,\rangle\!\rangle, 2) \Rightarrow_T^* (\langle\!\langle\, repeat(k){::}\, l,\ Bs\,\rangle\!\rangle, 2)$$

where *repeat*(k) in the final tape stands for a list of k consecutive pairs "B:1".

`Coq`'s proofs, that we have presented at a more abstract level, are very direct: apart from introducing the k variable to distinguish between an odd or even n, the user has just to carry out the actions indicated above (by using the constructors of the involved predicate) and manage the main inductive/coinductive proof principle. The only extra task is checking that the transition function behaves correctly (this is a novelty *w.r.t.* the proofs in previous sections), a goal which can be dealt with automatically by means of the `auto` tactic. □

Second example: R and L moves The second sample function that we choose is partially defined on input *pairs*, and may be named "partial minus":

$$pminus(m,n) \triangleq \begin{cases} m - n & \text{if } m \geq n \\ \uparrow & \text{if } m < n \end{cases}$$

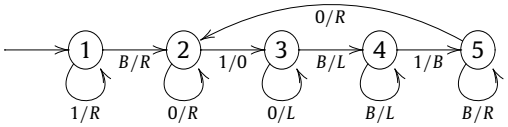
[11] Like discussed in Section 4, we display coinductive proofs, compactly, in a natural deduction-style: the coinductive hypothesis is indexed with the rule it is discharged by.

As assumed in Section 2, the two parameters are represented by the tape:

$$- \mid B \mid \underset{\Downarrow}{\underbrace{1 \mid - \mid 1}_{m+1}} \mid B \mid \underbrace{1 \mid - \mid 1}_{n+1} \mid B \mid - \tag{4}$$

formalized as $\langle\!\langle Bs,\ 1{:}ones(m){::}B{:}1{:}ones(n){::}Bs \rangle\!\rangle$ (*ones* as in previous example).

To compute *pminus*, we devise the following algorithm. First scan the tape towards the right till crossing the B that separates the two inputs; then erase the leftmost "1" from the representation of n and the rightmost "1" from that of m (both the "1s" must occur) by replacing them, respectively, with a mark symbol "0" (on the right, for n) and a B (on the left). The core of the computation is repeating this cycle, which leads to one of two possible cases: if the end of n is reached (*i.e.*, we are scanning the first B on the right of a 0-block), then stop; on the other hand, replacing m with B symbols may cause that the head (looking for "1s") moves indefinitely on the left. Its specification U is:



(1) starts the computation, by moving to the leftmost "1" of n and to state 2;
(2) keeps scanning a 0-block towards the right, moving to state 3 if a "1" is found and acting as the unique final state if a B if found (the n parameter has been completely "consumed" by means of 0-marks, hence $m \geq n$);
(3) scans a 0-block towards the left, moving to state 4 when the B that separates the representations of m and n is found;
(4) scans a B-block towards the left, either moving to state 5 if a "1" is eventually found, or moving forever to the left (m is "consumed", hence $m < n$);
(5) scans a B-block (between m and n) towards the right and moves to state 2 when a "0" is eventually found, thus starting the cyclic comparison process.

The initial part of the formal development (erasing the first pair of "1", so moving from state 1 to 5) is common to the divergent and convergent cases[12]:

$$- \mid B \mid \underset{\Downarrow}{\underbrace{1 \mid - \mid 1}_{m+1}} \mid B \mid \underbrace{1 \mid - \mid 1}_{n+1} \mid B \mid - \overset{*}{\underset{U}{\Longrightarrow}} - \mid B \mid 1^m \mid B \mid B \mid 0 \mid 1^n \mid B \mid - \qquad \Downarrow$$

and it is proved easily, by induction on m and via an obvious sequence of actions.

At this point, it is convenient to generalize the pattern of the tape as:

$$- \mid B \mid \underbrace{1 \mid - \mid 1}_{m} \mid \underbrace{B \mid - \mid B}_{k+2} \mid \underset{\Downarrow}{\underbrace{0 \mid - \mid 0}_{k+1}} \mid \underbrace{1 \mid - \mid 1}_{n} \mid B \mid - \tag{5}$$

Starting from this tape and state 5, we can discriminate between divergence and convergence by distinguishing the case $m < n$ from $m \geq n$. Notice that we have introduced the variable k to obtain a more powerful induction hypothesis.

We prove the *divergence* from the configuration given by the tape (5) and state 5, under the hypothesis $m < n$, by nested induction on n and m:

$$\forall\, n, m, k{\in}\mathbb{N}, \forall\, r{\in}HTape.$$
$$(m < n) \Rightarrow (\langle\!\langle\, blanks(k{+}2){::}ones(m){::}Bs,\ zeros(k{+}1){::}ones(n){::}r \,\rangle\!\rangle, 5) \Rightarrow^{\infty}_{U}$$

where the recursive functions *blanks* and *zeros* build a list of consecutive B and "0" symbols, similarly to *ones*, introduced in the first example of the section. The above proof requires auxiliary lemmas, to scan 0-blocks and B-blocks (by induction on k) and for assuring the divergence from the state 4 with the tape *Bs* from the head to the left. The key point is that we can use the predicate b_∞ in a *compositional* way: *i.e.*, when carrying out a divergence proof in a top-down fashion, we can perform a preliminary finite number of actions, thus reducing to a different goal. In fact, this amounts to splitting a divergent computation into a convergent one, easily provable, plus another divergent one, which becomes our goal. For example, we can scan a B-block (of length k) towards the right by applying (via the `apply` tactic) the lemma (proved by induction on k):

[12] Informally, we represent with $\overset{*}{\Longrightarrow}$ the effect of a finite number of actions on a tape. Moreover, we denote with 1^m a block of m consecutive squares that contain a "1" symbol.

$$\forall k{\in}\mathbb{N}, \forall l, r{\in}HTape.\ (\langle\!\langle\ blanks(k)::l,\ r\ \rangle\!\rangle, 5) \Rightarrow_U^{\infty} \Rightarrow (\langle\!\langle\ l,\ blanks(k)::r\ \rangle\!\rangle, 5) \Rightarrow_U^{\infty}$$

Conversely, it is *not* possible to use the predicate b_* in a compositional way to manage the *convergence* scenario. The problem is that b_* requires to exhibit the final tape, but in this case, due to the complexity of the proof, we cannot master it *tout-court* as we have done in the first example. Therefore, we need an extra tool to accomplish the convergence. Actually, such a tool is provided by the *small-step* predicate $\overset{*}{\rightarrow}$; by applying Lemma 1(3), we may decompose a convergent computation and address separately its intermediate steps:

$$\frac{(\langle\!\langle l, blanks(k)::r\rangle\!\rangle, 5) \overset{*}{\rightarrow}_U (\langle\!\langle blanks(k)::l, r\rangle\!\rangle, 5) \Rightarrow_U^* (\langle\!\langle l', r'\rangle\!\rangle, 2)}{(\langle\!\langle l, blanks(k)::r\rangle\!\rangle, 5) \Rightarrow_U^* (\langle\!\langle l', r'\rangle\!\rangle, 2)}\ \text{(Lemma 1(3))}$$

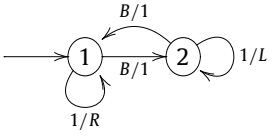
where l, r, l', r' are particular streams that occur in the convergence proof. Now, the lefthand reduction $(\langle\!\langle l, blanks(k)::r\rangle\!\rangle, 5) \overset{*}{\rightarrow}_U (\langle\!\langle blanks(k)::l, r\rangle\!\rangle, 5)$ can be proved readily (by induction on k), hence the righthand evaluation, leading the latter configuration to $(\langle\!\langle l', r'\rangle\!\rangle, 2)$, becomes the new (simpler) goal. In the end, we are able to carry out the convergence proof from (5) and state 5, under the hypothesis $m \geq n$, by nested induction on n and m, via lemmas similar to those used for b_∞, but this time involving the reduction relation, as explained above:

$$\forall n, m, k{\in}\mathbb{N}.\ (m \geq n) \Rightarrow (\langle\!\langle\ blanks(k{+}2)::ones(m)::Bs,\ zeros(k{+}1)::ones(n)::Bs\ \rangle\!\rangle, 5) \Rightarrow_U^* (\langle\!\langle\ zeros(k{+}1{+}n)::blanks(k{+}2{+}n)::ones(m{-}n)::Bs,\ Bs\ \rangle\!\rangle, 2)$$

Coq's proofs dealing with this second example are considerably more difficult than those addressing the first example. Essentially, this is caused by the complexity of the algorithm (*i.e.*, the resulting automata), which has to deal with two input parameters. The user is then forced to decompose the main computation in such a way that it is possible to devise sufficiently powerful inductive hypotheses to manage pieces of computations that occur inside the main proof. Moreover, the big-step convergence predicate, alone, cannot be exploited to accomplish the convergence case, as explained above. In the end, the proof requires the application of more than one hundred tactics. □

Third example: R and L moves, infinitely We consider now the unary function $f_\emptyset$, undefined on every input n, for which we devise an implementation that points out a limitation involving the mechanization of coinduction.

In fact, our algorithm to compute $f_\emptyset$ is tricky: first scan the 1-block towards the right and replace the first blank with a "1", then move the head towards the left till reaching the first blank and replace it again with a "1"; proceed infinitely in the same way. The specification V we pick out is minimal:



The idea beneath the formal divergence proof is nesting a couple of inductions inside the main coinduction; *i.e.*, by using the notation introduced in the previous example to display the modification of the tape, we want to perform the evaluations (moving from state 1 to 2 and then coming back to 1):

$$\underset{\Downarrow}{-\mid B \mid \underbrace{1 \mid - \mid 1}_{n+1} \mid B \mid -} \overset{*}{\underset{V}{\Longrightarrow}} \underset{\Downarrow}{-\mid B \mid \underbrace{1 \mid - \mid 1}_{n+2} \mid B \mid -} \overset{*}{\underset{V}{\Longrightarrow}} \underset{\Downarrow}{-\mid B \mid \underbrace{1 \mid - \mid 1}_{n+3} \mid B \mid -}$$

These two finite computations may be addressed via the two following lemmas:

Lemma scanR: $\forall k{\in}\mathbb{N}, \forall l{\in}HTape.\ (\langle\!\langle ones(k)::l,\ 1{:}Bs\rangle\!\rangle, 2) \Rightarrow_V^{\infty} \Rightarrow (\langle\!\langle l,\ ones(k)::Bs\rangle\!\rangle, 1) \Rightarrow_V^{\infty}$

Lemma scanL: $\forall k{\in}\mathbb{N}, \forall r{\in}HTape.\ (\langle\!\langle Bs,\ ones(k{+}2)::r\rangle\!\rangle, 1) \Rightarrow_V^{\infty} \Rightarrow (\langle\!\langle ones(k)::Bs,\ 1{:}r\rangle\!\rangle, 2) \Rightarrow_V^{\infty}$

that we prove by induction on k. Therefore, to accommodate the main divergence proof $\forall n{\in}\mathbb{N}.\ (\langle\!\langle Bs,\ ones(n{+}1)::Bs\rangle\!\rangle, 1) \Rightarrow_V^{\infty}$, we may assume such a goal as coinductive hypothesis and then apply, in turn, the two lemmas above, thus reducing to the configuration $(\langle\!\langle Bs,\ ones(n{+}3)::Bs\rangle\!\rangle, 1)$, which is an instance (for $n{+}2$, where n is assumed too) of the coinductive hypothesis itself:

$$\dfrac{\dfrac{\dfrac{[\forall n{\in}\mathbb{N}.\,(\langle\!\langle Bs,\ ones(n{+}1){::}Bs\rangle\!\rangle, 1) \Rightarrow_V^\infty]_{(1)}}{\vdots}}{(\langle\!\langle Bs,\ ones(n{+}3){::}Bs\rangle\!\rangle, 1) \Rightarrow_V^\infty}\ \text{(Lemma scanL)}}{(\langle\!\langle ones(n{+}1){::}Bs,\ 1{:}Bs\rangle\!\rangle, 2) \Rightarrow_V^\infty}\ \text{(Lemma scanR)}}{\forall n{\in}\mathbb{N}.\,(\langle\!\langle Bs,\ ones(n{+}1){::}Bs\rangle\!\rangle, 1) \Rightarrow_V^\infty}\ n{\in}\mathbb{N}\ (1), \text{(introduction)} \tag{6}$$

Nevertheless, it is apparent that the application of the coinductive hypothesis, which could conclude the proof, is *not* guarded by constructors, because no constructor of the b_∞ predicate is applied *at all* between the assumption of the coinductive hypothesis and its use. Instead, *two lemmas* ("scanR" and "scanL") are applied meanwhile: in fact, their application breaks the guard condition.[13]

To circumvent the problem, we introduce here a new small-step divergence predicate, equivalent to b_∞, to be used in a guarded way to accomplish the divergence proof (6). The idea is very direct: we want to split a divergent computation into a convergent one plus a divergent one by using a *constructor*; actually, this new divergence may be characterized as the *coinductive* transitive closure of the *inductive non-reflexive* transitive closure of $\rightarrow$.

Definition 3 *(Extra reduction).* Assume $T{\in}Spec$, $s, t, u{\in}Tape$, $p, q, i{\in}State$. Then, *finite positive* reduction $\overset{+}{\rightarrow}$ is defined by induction, via the rules[14]:

$$\dfrac{(s,p) \rightarrow_T (t,q)}{(s,p) \overset{+}{\rightarrow}_T (t,q)}\ (\overset{+}{\rightarrow}_1) \qquad \dfrac{(s,p) \rightarrow_T (u,i) \quad (u,i) \overset{+}{\rightarrow}_T (t,q)}{(s,p) \overset{+}{\rightarrow}_T (t,q)}\ (\overset{+}{\rightarrow}_+)$$

And *infinite guarded reduction* $\uparrow^\infty$ is defined by the following coinductive rule:

$$\dfrac{(s,p) \overset{+}{\rightarrow}_T (t,q) \quad (t,q) \uparrow_T^\infty}{(s,p) \uparrow_T^\infty}\ (\uparrow_\infty^\infty)$$

Lemma 2 *(Auxiliary). Assume $T{\in}Spec$, $s, t, u{\in}Tape$, and $p, q, i{\in}State$.*

1. *If $(s,p) \overset{+}{\rightarrow}_T (u,i)$ and $(u,i) \overset{+}{\rightarrow}_T (t,q)$, then $(s,p) \overset{+}{\rightarrow}_T (t,q)$*
2. *$(s,p) \uparrow_T^\infty$ if and only if $(s,p) \overset{\infty}{\rightarrow}_T$*

Proof. 1) By structural induction on the derivation of $(s,p) \overset{+}{\rightarrow}_T (u,i)$. 2) Both directions by coinduction and hypothesis inversion. □

Proposition 4 *(Equivalence, bis). Assume $T{\in}Spec$, $s{\in}Tape$, and $p{\in}State$.*

1. *$(s,p) \Rightarrow_T^\infty$ if and only if $(s,p) \uparrow_T^\infty$*

Proof. ($\Rightarrow$) By coinduction and hypothesis inversion. ($\Leftarrow$) By Proposition 3(2), part "if", and Lemma 2(2), part "only if". □

Since reduction $\uparrow^\infty$ is equivalent to evaluation b_∞, we adopt the former to carry out the divergence proof (6). Actually, $\uparrow^\infty$ does not suffer from the non-guardedness problem, because we use immediately its constructor ($\uparrow_\infty^\infty$) to split the proof into two subgoals, one of which can be solved immediately with the application of the coinductive hypothesis. This is apparent from the proof tree, where we write s for $\langle\!\langle Bs,\ ones(n{+}1){::}Bs\rangle\!\rangle$ and t for $\langle\!\langle Bs,\ ones(n{+}3){::}Bs\rangle\!\rangle$:

$$\dfrac{\dfrac{(s,1) \overset{+}{\rightarrow}_V (t,1) \qquad \dfrac{[\forall n{\in}\mathbb{N}.\,(s,1) \uparrow_V^\infty]_{(1)}}{\vdots}}{(t,1) \uparrow_V^\infty}}{(s,1) \uparrow_V^\infty}\ (\uparrow_\infty^\infty)}{\forall n{\in}\mathbb{N}.\,(s,1) \uparrow_V^\infty}\ n{\in}\mathbb{N}\ (1), \text{(introduction)}$$

The proof of the subgoal $(s,1) \overset{+}{\rightarrow}_V (t,1)$ relies on the transitivity of $\overset{+}{\rightarrow}$ (Lemma 2(1)) and on two auxiliary lemmas, analogous to the "scanR" and "scanL" ones, involving the predicate b_∞, that we have proved above. The reader can see that this

[13] See Section 4 for a thorough discussion about the implementation of coinduction in Coq.

[14] As usual, we write the instances of predicates, more suggestively, by using subscripts.

way of splitting a diverging computation is similar to the solution we had devised to deal with convergence in the second example of this section.

Once introduced the auxiliary diverging predicate $\uparrow^{\infty}$, the difficulty of Coq proofs is comparable to that experienced with the first example, therefore the same remarks apply. On the other hand, dealing formally with the extra reduction concepts (Lemma 2 and Proposition 4) requires an effort similar to addressing the equivalence between the semantics in Section 6. □

8. Join

In this section we address a completely new issue, namely the *sequential composition*, *a.k.a. join*, of TMs. Such an operation allows one to define the semantics of a machine via the semantics of its submachines, thus achieving a first level of abstraction to reason about TMs. From the opposite perspective, one can analyze the single machines independently from each other, and then "compose" both the machines and their semantics to address the properties of the whole "structured" machine. We will point out the potential of the join by proving the undecidability of the halting problem, in the next section.

Precisely, the join operation can be defined on the *specification* component of machines, without considering their (independent) tape component. The formalization we have chosen for specifications, *i.e.*, lists of quadruples in *Spec* (Section 3), offers us the possibility of a very direct implementation of the join, which we realize simply as the *append* function on lists. After presenting the technical details, we will discuss thoroughly the motivations of our choice.

In fact, we write "$M; N$" for the join of the specifications M and N, with the intended meaning that evaluating such a specification means evaluating M, with its input tape, until it stops (if this is the case) and then evaluating N with M's output tape. Therefore, we look for a side-condition to get a couple of "rules" for decomposing converging and diverging evaluations as follows[15]:

$$\frac{(s,p) \Rightarrow^{*}_{M} (t,q) \quad (t,q) \Rightarrow^{*}_{N} (u,r)}{(s,p) \Rightarrow^{*}_{M;N} (u,r)}\ (*) \qquad \frac{(s,p) \Rightarrow^{*}_{M} (t,q) \quad (t,q) \Rightarrow^{\infty}_{N}}{(s,p) \Rightarrow^{\infty}_{M;N}}\ (\infty)$$

In order to make valid these rules, we require the *target* states belonging to N to be *disjoint* from the *source* states in M, a constraint which is actually taken at a bare minimum and that we write $M_S \cap N_T = \emptyset$. The motivation of such a side-condition relies on the fact that, when the control is got by the second specification, it must not come back to the first one. We address formally this characterization via the following statements.

Lemma 3 *(Transitions). Assume $M, N \in Spec$, $s, t \in Tape$, $p, q, i \in State$, $x \in Head$.*

1. *If $tr(M,p,s)\downarrow$, then $tr(N,p,s) = tr(M;N,p,s)$*
2. *If $tr(M,p,s) \mapsto \langle i,x \rangle$, then $tr(M,p,s) = tr(M;N,p,s)$*
3. *If $tr(M,p,s)\downarrow$, $(s,p) \Rightarrow^{*}_{N} (t,q)$, and $M_S \cap N_T = \emptyset$, then $(s,p) \Rightarrow^{*}_{M;N} (t,q)$*
4. *If $tr(M,p,s)\downarrow$, $(s,p) \Rightarrow^{\infty}_{N}$, and $M_S \cap N_T = \emptyset$, then $(s,p) \Rightarrow^{\infty}_{M;N}$*

Proof. 1) and 2) By induction on the structure of M. 3) By induction on the structure of the derivation $(s,p) \Rightarrow^{*}_{N} (t,q)$ and point (1). 4) By coinduction, inversion of the hypothesis $(s,p) \Rightarrow^{\infty}_{N}$ and point (1). □

Proposition 5 *(Join). Assume $M, N \in Spec$ such that $M_S \cap N_T = \emptyset$, $s, t, u \in Tape$, and $p, q, r \in State$.*

1. *If $(s,p) \Rightarrow^{*}_{M} (t,q)$ and $(t,q) \Rightarrow^{*}_{N} (u,r)$, then $(s,p) \Rightarrow^{*}_{M;N} (u,r)$*
2. *If $(s,p) \Rightarrow^{*}_{M} (t,q)$ and $(t,q) \Rightarrow^{\infty}_{N}$, then $(s,p) \Rightarrow^{\infty}_{M;N}$*

Proof. Both the points need the application of Lemma 3(2). 1) By induction on the structure of the derivation $(s,p) \Rightarrow^{*}_{M} (t,q)$ and Lemma 3(3). 2) By coinduction, inversion of the hypothesis $(s,p\rangle \Rightarrow^{*}_{M} (t,q\rangle$ and Lemma 3(4). □

This Proposition provides us with the formal version of the two above rules $(*)$, (∞): given two machines M and N (we will keep saying "machines" in place of "specifications", which is a little abuse of terminology), if the source states of M cannot be reached from N (side-condition $M_S \cap N_T = \emptyset$), it is possible to derive the semantics of the join machine "$M; N$" from the semantics of the two "brick" machines M and N, independently from the tape component.

Coq's proofs in this section are not difficult, but a little laborious. When they are carried out by induction on specifications, we must inspect them and compare states and alphabet symbols. Also managing the join side-condition within proofs performed by (co)induction on derivations is an extra effort.

[15] Notice that we do not introduce a third rule, where the lefthand evaluation (*i.e.*, the one involving M) diverges, because it would be useless in practice for our objectives.

Our implementation of the join by means of Coq's built-in *append* function is a direct consequence of the choices made previously. We have modeled specifications as lists, and transitions via a recursive function that traverses such lists from left to right. Actually, our formalization is minimal, with no extra constraints about initial and final states and about the length of specifications. Such a minimality is therefore preserved by the join definition.

Obviously, when working with concrete TMs (as in the next section), we will have to rearrange their states and transitions to respect the join side-condition.

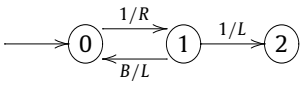
9. The halting problem

As an application of the methodology introduced in the previous section (sequential composition of TMs), we prove here the *undecidability of the halting problem*, by following the proof argument developed by Boolos and coworkers in the textbook [2], that we instantiate in our mechanized environment.

To simplify the proof of the goal, we tune our machinery by taking state 0 as the initial one and working with machines that stop at their highest state; for a machine M, we denote such a state by M_j. We postpone to the end of the section a discussion that our assumption is without loss of generality.

The proof of undecidability is accomplished in [2] via the introduction of two concrete machines, that we now certify in isolation and then join together.

Dithering machine The first is a trivial machine that checks its input parameter, stopping if it represents a number greater than zero and looping otherwise. Its specification is the simplest one whose certification we address in this paper (compare with Section 7); in particular, we remark that it loops "in place", *i.e.*, without scrutinizing an infinite amount of tape. The machine D is defined as:



We evaluate D on input $1 \in \mathbb{N}$ for convergence, on input $0 \in \mathbb{N}$ for divergence:

$$\begin{array}{cc} \Downarrow & \Downarrow \\ - \mid B \mid 1 \mid 1 \mid B \mid - & - \mid B \mid 1 \mid B \mid - \end{array}$$

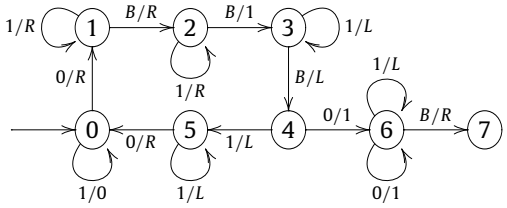
State 0 starts the computation (the first "1" is mandatory), moving to state 1. State 1 discriminates between an input equal to or greater than $0 \in \mathbb{N}$: in the latter case it moves to state 2 (a B symbol is found), otherwise it moves back to state 0, thus leading to a looping pair of moves between the two states. State 2 behaves as the unique final state (no transition out of it is provided).

The proof of convergence, $(\langle\!\langle Bs,\ 1{:}1{:}Bs \rangle\!\rangle, 0) \Rightarrow_D^* (\langle\!\langle Bs,\ 1{:}1{:}Bs \rangle\!\rangle, 2)$, is immediate via one R and one L-move, while the divergence, $(\langle\!\langle Bs,\ 1{:}Bs \rangle\!\rangle, 0) \Rightarrow_D^\infty$, is proved by straightforward coinduction on the derivation, again by means of a pair of R and L-moves, which this time are repeated "ad infinitum". □

Copying machine The second machine C has the purpose of making a copy of a parameter n, by transforming the tape representing it as follows:

$$\begin{array}{ccc} \Downarrow & & \Downarrow \\ - \mid B \mid 1 \mid 1^n \mid B \mid - & \overset{*}{\Longrightarrow}_C & - \mid B \mid 1 \mid 1^n \mid B \mid 1 \mid 1^n \mid B \mid - \end{array}$$

The algorithm we pick out is implemented by the following specification C:



State 0 works within the lefthand 1-block, by replacing the leftmost occurrence of a "1" symbol with a "0" and then moving to state 1. Such a state scans the lefthand 1-block to the right, till crossing the first B, when it moves to state 2. State 2 scans the righthand 1-block to the right, till reaching the first B, that it replaces with a "1" by moving to state 3. Such a state scans the righthand 1-block to the left, till crossing the middle B, when it moves to state 4. State 4 discriminates between two cases: either the *loop* part is complete (therefore it starts replacing "0s" with "1s" and moves to state 6) or a new cycle has to begin (it starts scanning the lefthand 1-block to the left by moving to state 5). Such a state scans the lefthand 1-block to the left, till reaching a "0", when it enters state 0 and moves the head to the right. State 6 replaces all

the "0s" with "1s" and moves to state 7 by finalizing the position of the head. State 7 acts as the unique final state (no transition out of it is provided).

The proof that C carries out its goal is split in three parts. In the first (*begin* part), replace the leftmost occurrence of "1" (at least one must exist) by a "0"; scan the tape on the right till to a second B and replace it with a "1"; move the head to the left to scan the middle B (the state transition is from 0 to 3):

$$\begin{array}{c} \Downarrow \\ - \mid B \mid 1 \mid 1^n \mid B \mid - \end{array} \overset{*}{\Longrightarrow}_C \begin{array}{c} \Downarrow \\ - \mid B \mid 0 \mid 1^n \mid B \mid 1 \mid B \mid - \end{array}$$

The second part of the proof (named *loop*) addresses the core cycle: move left, replacing the leftmost "1" (if any, *i.e.*, $n > 0$) with a "0"; correspondingly, replace the first B on the right of the righthand 1-block with a "1"; move left to scan the middle B. This part of the proof requires generalizing its pattern (a solution similar to, but easier than, the second example of Section 7):

$$\begin{array}{c} \Downarrow \\ - \mid B \mid 0^{k+1} \mid 1^i \mid B \mid 1^{k+1} \mid B \mid - \end{array} \overset{*}{\Longrightarrow}_C \begin{array}{c} \Downarrow \\ - \mid B \mid 0^{n+1} \mid B \mid 1^{n+1} \mid B \mid - \end{array}$$

Hence, the *loop* part can be mastered by proving (by induction on i, under the hypothesis $n = i+k$) that evaluating C updates the tape in such a way (the target and source states coincide, *i.e.*, 3) and then instantiating it with $k = 0$.

The last fragment of the proof (*end* part) consists in moving the head towards the left and replacing all the "0s" with "1s" (state transition from 3 to 7):

$$\begin{array}{c} \Downarrow \\ - \mid B \mid 0^{n+1} \mid B \mid 1^{n+1} \mid B \mid - \end{array} \overset{*}{\Longrightarrow}_C \begin{array}{c} \Downarrow \\ - \mid B \mid 1 \mid 1^n \mid B \mid 1^{n+1} \mid B \mid - \end{array}$$

The three points addressed above allow us to prove the main goal, *i.e.*, that the machine C stops by completing the "copy" of its input parameter:

$$\forall n \in \mathbb{N}.\ (\langle\!\langle Bs,\ 1^{n+1}{::}Bs \rangle\!\rangle, 0) \Rightarrow_C^* (\langle\!\langle Bs,\ 1^{n+1}{::}B{:}1^{n+1}{::}Bs \rangle\!\rangle, 7)$$

We remark that in the proofs of the *begin*, *loop*, *end* parts we have exploited some auxiliary lemmas to scan blocks of "1s" (by induction on their length), while the only extra tool is performing head moves to connect the configurations.

We also add that we have not designed the machine C as a join of submachines, like in [13], for minimality and simplicity reasons, as the *loop* part reuses all the states and transitions of the *begin* part (in other words, in the present case one should introduce more states and transitions, thus complicating the formal proof, to work with the join). We stress that we are going to exploit the join operation in the rest of the section, though. □

Undecidability of the halting problem In Turing setting, the *halting problem* amounts to finding a machine H such that, when given as input any machine M, represented by its *code number* $m \in \mathbb{N}$, and a parameter n, produces as output the answer to the question whether M halts or not on input n. The function computed by such a hypothetical halting *machine* H would be the halting *function* $h(m, n)$, which outputs the values 1 for halt and 2 otherwise.[16]

In [2], the core of the proof that this problem is not solvable consists in constructing, by means of H, a witness machine W. Then, W is run on its code number; both supposing that W halts and supposing the opposite lead to a contradiction: the conclusion is that such a machine H *cannot exist*.

Since in our framework the functions computed by TMs are not, at the moment, first-class citizens (*i.e.*, they cannot be handled at metalanguage level), we have to address the undecidability proof in a slight different way.

Our approach is closely related to that of Xu, Zhang and Urban, performed in `Isabelle/HOL`[13], with differences motivated by our formalization setting. Our version of the demonstration is based on the following preliminaries.

1. First, we postulate the existence of a function γ, which returns the code number of a machine M (no further assumptions are needed about γ).
2. Second, we define a predicate $halt \subseteq Spec \times \mathbb{N}$, that characterizes the way a machine M either stops or not, when run on input n:

$$halt(M, n) \triangleq \exists t \in Tape.\ (\langle\!\langle Bs,\ 1^{n+1}{::}Bs \rangle\!\rangle, 0) \Rightarrow_M^* (t, M_j) \tag{7}$$

   where, as assumed at the outset of the section, M_j is M's highest state.
3. Finally, we assume the existence of a halting machine H, which computes the function $h\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ envisaged above. In fact, H would be capable of telling whether a machine M with code number m either halts or not on input n. Therefore, we can state the following logical connection between the evaluations of H and the predicate *halt*, defined at previous point:

[16] Notice that it is not necessary to reason about *functions* in this particular proof, though.

$$\begin{array}{lcl} halt(M,n) & \Rightarrow & (t,0) \Rightarrow_H^* (\langle\!\langle Bs,\ 1{::}Bs\rangle\!\rangle, H_j) \\ \neg halt(M,n) & \Rightarrow & (t,0) \Rightarrow_H^* (\langle\!\langle Bs,\ 1{:}1{::}Bs\rangle\!\rangle, H_j) \end{array} \tag{8}$$

where the input tape $t \triangleq \langle\!\langle Bs,\ 1^{m+1}{::}B{:}1^{n+1}{::}Bs\rangle\!\rangle$ represents the *input* parameters m and n, and the target tapes encode the *output* 1 and $2 \in \mathbb{N}$.

We encode in Coq the function γ and the halting machine H as follows:

```
Parameter gamma: Spec -> nat. Parameter H: Spec.
```

The *witness* machine is built via the C (copying), the hypothetical H (halting) and the D (dithering) machines, which are joined together. The idea is to run C on its input parameter, thus making a copy of it. Afterwards, H is evaluated starting from the resulting tape, which means that we can interpret its first parameter as a code number of a certain machine: hence, the objective is to address the convergence and the divergence of such a machine through the implications (8). Finally, the tape arising from the finite computation under H is given as input to D, which may then either stop or loop.

It is now time to define the witness machine W; to join safely the three brick machines, we must respect the side-condition devised in the previous section:

$$W \triangleq C\,;\, shift(H,7)\,;\, shift(D, H_j{+}7)$$

Actually, *shift*: $Spec \times \mathbb{N} \to Spec$ is a function that "adds" an amount (its second parameter) to the transitions of a machine (and also to its states), thus allowing the involved machine to be successfully composed with the rest of the system. It is immediate that such an operation preserves the semantics.

Lemma 4 *(Shift). Assume $T \in Spec$, $s, t \in Tape$, $p, q, i \in State$, $x \in Head$, and $n \in \mathbb{N}$.*

1. *If $tr(T,p,s)\downarrow$, then $tr(shift(T,n), p{+}n, s)\downarrow$*
2. *If $tr(T,p,s) \mapsto \langle i, x\rangle$, then $tr(shift(T,n), p{+}n, s) \mapsto \langle i{+}n, x\rangle$*
3. *If $(s,p) \Rightarrow_T^* (t,q)$, then $(s, p{+}n) \Rightarrow_{shift(T,n)}^* (t, q{+}n)$*
4. *If $(s,p) \Rightarrow_T^\infty$, then $(s, p{+}n) \Rightarrow_{shift(T,n)}^\infty$*

Proof. 1) and 2) By induction on the structure of the specification T. 3) By induction on the structure of the hypothetical derivation, and points (1) and (2). 4) By coinduction, inversion of the hypothesis and point (2). □

The undecidability proof is argued by *classical* logic; we evaluate the machine W on its code number $\gamma(W) = m$ and then proceed by case analysis.

- First, we suppose that $halt(W,m)$, which, according to Definition (7), means $(\langle\!\langle Bs,\ 1^{m+1}{::}Bs\rangle\!\rangle, 0) \Rightarrow_W^* (t, W_j)$ for some tape t; hence, we can deduce $\neg(\langle\!\langle Bs,\ 1^{m+1}{::}Bs\rangle\!\rangle, 0) \Rightarrow_W^\infty$ via Proposition 1(1). However, by exploiting the first implication of (8), we can also build a derivation of $(\langle\!\langle Bs,\ 1^{m+1}{::}Bs\rangle\!\rangle, 0) \Rightarrow_W^\infty$, which leads immediately to inconsistency.
- On the other hand, assuming $\neg halt(W,m)$ allows us to derive, via the second implication of (8), $(\langle\!\langle Bs,\ 1^{m+1}{::}Bs\rangle\!\rangle, 0) \Rightarrow_W^* (\langle\!\langle Bs,\ 1{:}1{:}Bs\rangle\!\rangle, W_j)$, *i.e.*, according to (7), $halt(W,m)$, which is again a contradiction.

In detail, the structure of the derivation of $(s,0) \Rightarrow_W^\infty$ (first path) is as follows:

$$\dfrac{(s,0) \Rightarrow_C^* (t,7) \qquad \dfrac{(t,7) \Rightarrow_{H'}^* (u, H_j{+}7) \quad (u, H_j{+}7) \Rightarrow_{D'}^\infty}{(t,7) \Rightarrow_{H';D'}^\infty}\ \text{(Proposition 5(2))}}{(s,0) \Rightarrow_W^\infty}\ \text{(Proposition 5(2))}$$

where $H'{=}shift(H,7)$, $D'{=}shift(D, H_j{+}7)$, and the tapes have appeared in the section: $s{=}\langle\!\langle Bs,\ 1^{m+1}{::}Bs\rangle\!\rangle$, $t{=}\langle\!\langle Bs,\ 1^{m+1}{::}B{:}1^{m+1}{::}Bs\rangle\!\rangle$, $u{=}\langle\!\langle Bs,\ 1{:}Bs\rangle\!\rangle$.

If $v{=}\langle\!\langle Bs,\ 1{:}1{:}Bs\rangle\!\rangle$, the derivation of $(s,0) \Rightarrow_W^* (v, W_j)$ (second path) is:

$$\dfrac{(s,0) \Rightarrow_C^* (t,7) \qquad \dfrac{(t,7) \Rightarrow_{H'}^* (v, H_j{+}7) \quad (v, H_j{+}7) \Rightarrow_{D'}^* (v, W_j)}{(t,7) \Rightarrow_{H';D'}^* (v, W_j)}\ \text{(Proposition 5(1))}}{(s,0) \Rightarrow_W^* (v, W_j)}\ \text{(Proposition 5(1))}$$

Auxiliary tools used in both paths of the proof concern the *shift* function (Lemma 4, points (3), (4)), while the second path requires also $W_j = H_j{+}9$. Concerning Coq's proofs, in this section no new particular difficulty arises. □

Table 1
Size of the formalization.

Big-step semantics	174 lines	4.66 KB
Adequacy (lists vs. streams)	596 lines	16.85 KB
Adequacy (big vs. small-step)	163 lines	4.03 KB
Examples (Section 7)	840 lines	21.44 KB
Join	390 lines	8.86 KB
Shift	203 lines	5.20 KB
Dither machine	30 lines	0.80 KB
Copy machine	301 lines	9.87 KB
Undecidability	302 lines	7.97 KB

Soundness As observed at the beginning of the section, the undecidability proof becomes simpler by working with machines that start at state 0 and stop at their highest state. In a former attempt, we experimented with machines starting at state 1 and stopping at 0, with the consequence that we had to introduce two extra machines, to join C with H and the resulting machine with D.

The fact that our "simplifying" assumption is without loss of generality relies on the following consideration. In our setting, totally-defined machines (with transitions for any pair state-symbol) could not stop. Therefore, any terminating machine, such as the halting H, must have at least one "gap", *i.e.*, a state-symbol configuration for which no transition is provided. To make these machines invariably halt at their highest state, it would be sufficient to extend them as follows: for any stopping configuration, add an action to move to a new, highest state by writing on the tape the same symbol already scanned by the head; finally, do not provide the highest state with any transition out of it. We conjecture that some version of such a property could be formalized in `Coq`.

10. Conclusion

In the present contribution we have formalized TMs and their (big-step and small-step) operational semantics in the `Coq` proof assistant. Our key choices are the encoding of tapes as pairs of *streams* (managed by means of corecursion) and a clear distinction between *converging* computations (modeled via inductive predicates) and *diverging* ones (formalized through coinductive predicates). In the core Sections 7, 8, and 9 we have pointed out the potential of our machinery, by proving the correctness of representative TMs (that is, by certifying the implementation of the partial functions computed by them), by introducing the sequential composition of TMs (which provides a "structuration" tool), and performing the proof of the undecidability of the halting problem, respectively.

Our encoding provides a completely mechanized management of the transitions (via the `auto` tactic), with the benefit that we may concentrate on the formal treatment of the tape and the logic of proofs. *Divergence* can be proved very often in a compositional way,[17] solely via the big-step coinductive predicate. When "non-guardedness" complications arise (due to `Coq`'s limited form of coinduction), alternative, equivalent small-step coinductive predicates may be employed, by taking advantage of their close relationship with the main big-step predicate. On the other hand, it is not always possible to master *convergence* proofs by compositionality. When this is not feasible (due to the difficulty of the proof at hand), the small-step semantics predicates may be used again as an auxiliary tool, to perform intermediate computation steps.

We note also that often, in order to carry out either divergence or convergence proofs, the user has the responsibility to figure out how to decompose the main goal. As usual, it is sometimes necessary to generalize the statements to obtain sufficiently powerful (co)inductive hypotheses. Moreover, some proofs require a subtle combination of inductive and coinductive reasoning.

We report in Table 1 a rough measure of the formalization, where we display the number of lines and the size of the scripts (not optimized and including comments) for the different parts of our work. As one can see, a major effort is necessary to establish the relationship with the semantics that use lists, while the proofs for the individual machines considered in this paper are a few hundred lines long. The overall dimension of the script is approximately 80 KB and it takes 16 seconds to proof check it by a `Coq` version 8.4pl5 that runs on a machine with two processors at 2 GHz and 4 GB RAM under the Windows 7 operating system. We refer the interested reader to the web appendix of this paper [5] for a more precise feedback about the details of the formalization.

As far as *related work* is concerned, the contributions most related to the present one are those by Asperti and Ricciotti in `Matita` [1], Xu, Zhang and Urban in `Isabelle/HOL` [13], and Leroy in `Coq` [10]. Both the first two works address TMs, achieving the ambitious goals we have mentioned in Section 1. We present now the contents of those papers most connected with our work.

TMs in `Isabelle/HOL` In [13], the authors start from the textbook [2], where tapes with just the two symbols "blank" and "occupied" are considered. They represent such tapes via a pair of *lists* (managed as a zipper) and add an *extra* "no-operation" action to the write and move ones. States are natural numbers and programs (namely, specifications, in our

[17] "Compositionality" concerns *computations*, whereas "sequential composition" *machines*.

terminology) contain actions for every pair "state-symbol", apart from the halting state 0. Hence, the transition function is *totally defined*, with a no-operation edge from the state 0 to itself.

The semantics is defined by means of an evaluation *function*, conceived to carry out exactly the number of steps indicated by one of its parameters (when needed, this is accomplished via no-operation steps from the state 0 to itself). A natural number n is represented by a cluster of $n+1$ "occupied" symbols.

The certification of TMs is approached via the concepts of *total correctness*, formalized through *Hoare-triples*, and *non-termination*, defined via *Hoare-pairs*.

Then, the sequential composition of TMs is introduced, by means of a machinery that deals appropriately with the redirection of transitions. Correspondingly, Hoare-rules are derived to reason about systems built by sequentially composed machines: these rules allow one to split the proof of correctness for the whole system into proofs that manage each machine separately from each other. Via such a machinery, the undecidability of the halting problem is proved.

TMs in `Matita` In [1], Asperti and Ricciotti motivate their choice for a symmetric representation of the tape (by means of a *triple*, made of two lists plus the symbol currently scrutinized), whose squares contain symbols from a finite-set parameter alphabet. Like in the contribution discussed before, the authors add a "no-operation" move of the head; in particular, TMs are allowed both to write and move in the same action. States are encoded again via a finite-set type, and among them there is a start state and a set of halting ones.

The semantics is approached by means of a computation function, which *iterates* a transition step between state-tape pairs until a final state is reached. If this is not the case, the computation is stopped after an upper bound number of iterations, and an optional value is returned to manage the non-termination.

Then, the semantics itself is defined through a *relation* between the input and the final *tapes*. In detail, a machine is said to *realize* a relation when, for each initial and final tapes that are in the relation, there exists a computation from the initial to the final tape (coupled to the starting state and a final one, obviously). It turns out that *termination* follows from realizability, and that termination plus *weak realizability* (requiring that input and output tapes are in relation provided there is a computation between them) imply realizability.

Coinductive semantics in `Coq` None of the above two works makes use of coinductive tools (that we have exploited to deal with stream-tapes and divergence); from this perspective, our paper is more related to that of Leroy [10], who adopts coinduction in `Coq` to describe diverging evaluations of a call-by-value λ-calculus.

The motivation of that work is to use coinduction for dealing with the *big-step* semantics, because such a semantics is more convenient than the small-step one to prove the correctness of *program transformations*, such as compilation.

Discussion In this work we have proposed an innovative approach to the formalization of the infinite structures that occur in TMs. Our effort may thus be seen as a *new* initial step towards the development of basic computability theory. We state now some considerations about the relationship between the present paper and the alternative contributions that formalize TMs.

In our opinion, the use of coinductive predicates for managing the divergence of computations is natural and elegant; we believe that this solution makes the description of the non-termination in a sense more "primitive" and points out a sort of duality with the convergence case. Also the choice of corecursion for modeling the tape is advantageous, as it allows us to rule out immediately the need of defining an equivalence relation on tapes (in [1], the reason for representing the tape via a triple is to avoid defining such an equivalence).

Actually, our approach is more similar to that carried out in `Isabelle/HOL` [13]. Up to date, our formalization has not required us to introduce extra concepts such as the well-formedness of specifications, the standard-form of tapes and the no-operation action (that are used in [13]), but we do not exclude the need to define them for pursuing more involved objectives. Concerning the certification of machines, it seems more difficult to reason via Hoare-triples (for total correctness) and Hoare-pairs (for non-termination) than via our predicates, because the former require one to exhibit invariants for any state and, in the case of termination, to pick out an order and a measure function. More similarities between the two approaches arise when we have to figure out a pattern for devising a sufficiently powerful (co)inductive hypothesis. As far as the sequential composition of machines, there is a strict correspondence between the Hoare-rules used in [13] and our properties defined in Proposition 5.

Future work We believe that the main result achieved by our contribution (*i.e.*, the development of a technology for proving the correctness of concrete TMs, via several versions of big-step and small-step semantics, with an application to the undecidability of the halting problem) is a promising tool to pursue more advanced goals which are outside the scope of the present paper.

In particular, our effort may be seen as a first step towards the development of computability theory, as the construction of "brick" TMs and their composition at higher-levels of abstraction is the natural progress of this contribution.

It would be also stimulating to relate the present formalization to that of unlimited register machines, that we have addressed in a previous work [3].

Acknowledgements

The author is very grateful to the anonymous referees for their helpful, constructive reviews.

References

[1] A. Asperti, W. Ricciotti, Formalizing Turing Machines, in: Proc. of WoLLIC, in: Lecture Notes in Computer Science, vol. 7456, Springer, 2012, pp. 1–25.
[2] G. Boolos, J. Burgess, R. Jeffrey, Computability and Logic, Cambridge University Press, 2007.
[3] A. Ciaffaglione, A coinductive semantics of the unlimited register machine, in: Proc. of INFINITY, in: Electronic Proceedings in Theoretical Computer Science, vol. 73, 2011, pp. 49–63.
[4] A. Ciaffaglione, A coinductive animation of Turing Machines, in: Proc. of SBMF, in: Lecture Notes in Computer Science, vol. 8941, Springer, 2014, pp. 80–95.
[5] A. Ciaffaglione, The Web Appendix of this paper, available at https://users.dimi.uniud.it/~alberto.ciaffaglione/Turing/Halting/.
[6] T. Coquand, Infinite objects in type theory, in: Proc. of TYPES, in: Lecture Notes in Computer Science, vol. 806, Springer, 1993, pp. 62–78.
[7] E. Giménez, Codifying guarded definitions with recursive schemes, in: Proc. of TYPES, in: Lecture Notes in Computer Science, vol. 996, Springer, 1994, pp. 39–59.
[8] N.J. Cutland, Computability: An Introduction to Recursive Function Theory, Cambridge University Press, 1980.
[9] C.-K. Hur, G. Neis, D. Dreyer, V. Vafeiadis, The power of parameterization in coinductive proof, in: Proc. of POPL, ACM, 2013, pp. 193–206.
[10] X. Leroy, Coinductive big-step operational semantics, in: Proc. of ESOP, in: Lecture Notes in Computer Science, vol. 3924, Springer, 2006, pp. 54–68.
[11] M. Norrish, Mechanised computability theory, in: Proc. of ITP, in: Lecture Notes in Computer Science, vol. 6898, Springer, 2011, pp. 297–311.
[12] A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, in: Proceedings of the London Mathematical Society, vol. 42, 1936.
[13] J. Xu, X. Zhang, C. Urban, Mechanising Turing Machines and computability theory in Isabelle/HOL, in: Proc. of ITP, in: Lecture Notes in Computer Science, vol. 7998, Springer, 2013, pp. 147–162.
[14] The Coq Development Team, The Coq Proof Assistant, version 8.4, INRIA, available at http://coq.inria.fr.