Formalizing Turing Machines

Andrea Asperti, Wilmer Ricciotti

Department of Computer Science, University of Bologna
asperti@cs.unibo.it, ricciott@cs.unibo.it

Abstract. We discuss the formalization, in the Matita Theorem Prover, of a few, basic results on Turing Machines, up to the existence of a (certified) Universal Machine. The work is meant to be a preliminary step towards the creation of a formal repository in Complexity Theory, and is a small piece in our Reverse Complexity program, aiming to a comfortable, machine independent axiomatization of the field.

1 Introduction

We have assisted, in recent years, to remarkable achievements obtained by means of interactive theorem provers for the formalization and automatic checking of complex results in many different domains, spanning from pure mathematics [9, 14, 5] to software verification [19, 18, 25, 1], passing through the metatheory and semantics of programming languages [10, 24].

Surprisingly, however, very little work has been done so far in major fields of theoretical computer science, such as computability theory and, especially, complexity theory. The only work we are aware of is [20], containing basic results in computability theory relying on λ-calculus and recursive functions as computational models. The computational constructs of both these models are not finitistic and are not very suitable for complexity purposes: Turing Machines still provide the standard foundation for this discipline.

Our work is an initial, preliminary contribution in this direction. In particular, we present a formalization of basic definitions and results on Turing Machines, up to the existence of a universal machine and the proof of its correctness. In particular, in Section 2 we discuss the notion of Turing Machine and its semantics; Section 3 provides means for composing machines (sequential composition, conditionals and iteration); Section 4 contains the definition of basic, atomic machines for textual manipulation of the tape; Section 5 introduces the notion of Normal Turing Machine and its standard representation as a list of tuples; Section 6 gives an outline of the universal machine; Section 7 and 8 are respectively devoted to the two main routines of the universal machine, namely *finding* the right tuple to apply, and *executing* the corresponding action; in Section 10, we summarize the main results which have been proved about the universal machine. In the conclusion we provide overall information about the size of the contribution and the resources required for its development as well as more motivations for pursuing formalization in computability and complexity theory: in particular we shall briefly outline our long term *Reverse Complexity*

program, aiming to a trusted, comfortable, machine independent axiomatization of the field suitable for mechanization.

In our development, we have been inspired by several traditional articles and textbooks, comprising e.g. [13, 17, 11, 23, 21]; however, it is worth to remark that none of them provides a description of the topic, and especially of universal machines, sufficiently accurate to be directly used as a guideline for formalization.

The formalization work described in this paper has been performed by means of the Matita Interactive Theorem Prover [8]. For lack of space we cannot provide details about proofs; the development will be part of the standard library of Matita since the next public release, and in the next few months will be made accessible on-line through the new Web interface of the system [6].

2 The notion of Turing Machine

Turing Machines were defined by Alan M. Turing in [22]. To Computer Scientists, they are a very familiar notion, so we shall address straight away their formal definition. Let us just say that, for the purposes of this paper, we shall stick to deterministic, single tape Turing Machines. The generalization to multi-tape/non deterministic machines does not look problematic.[1]

2.1 The tape

The first problem is the definition of the tape. The natural idea is to formalize it as a zipper, that is a pair of lists l and r, respectively representing the portions of the tape at the left and the right of the tape head; by convention, we may assume the head is reading the first symbol on the right. Of course, the machine must be aware this list can be empty, that means that the transition function should accept an *optional* tape symbol as input. Unfortunately, in this way, the machine is only able to properly react to a right overflow; the problem arises when the left tape is empty and the head is moved to the left: a new "blank" symbol should be added to the right tape. A common solution in textbooks is to reserve a special blank character $\sqcup$ of the tape alphabet for this purpose: the annoying consequence is that tape equality should be defined only up to a suitable equivalence relation ignoring blanks. To make an example, suppose we move the head to the left and then back to the right: we expect the tape to end up in the same situation we started with. However, if the tape was in the configuration $([\,], r)$ we would end up in $([\sqcup], r)$. As anybody with some experience in interactive proving knows very well, reasoning up to equivalence relations is

[1] It is worth to recall that the choice about the number of tapes, while irrelevant for computability issues, it is not from the point of view of complexity. Hartmanis and Stearns [15] have shown that any k-tape machine can be simulated by a one-tape machine with at most a quadratic slow-down, and Hennie [16] proved that in some cases this is the best we can expect; Hennie and Stearns provided an efficient simulation of multi-tape machines on a two-tape machine with just a logarithmic slow-down [12].

extremely annoying, that prompts us to look for a different representation of the tape.

The main source of our problem was the asymmetric management of the left and right tape, with the arbitrary assumption that the head symbol was part of the right tape. If we try to have a more symmetric representation we must clearly separate the head symbol from the left and right tape, leading to a configuration of the kind (l, c, r) (mid-tape); if we have no c, this may happen for three different reasons: we are on the left end of a non-empty tape (left overflow), we are on the right end of a non-empty tape (right overflow), or the tape is completely empty.

This definition of the tape may seem conspicuous at first glance, but it resulted to be quite convenient.

```
inductive tape (sig:FinSet) : Type :=
| niltape : tape sig
| leftof  : sig → list sig → tape sig
| rightof : sig → list sig → tape sig
| midtape : list sig → sig → list sig → tape sig.
```

For instance, suppose to be in a configuration with an empty left tape, that is (*midtape* [] *a l*); moving to the left will result in (*leftof a l*); further moves to the left are forbidden (unless we write a character to the uninitialized cell, therefore turning the overflow into a mid-tape), and moving back to the right restores the original situation.

Given a tape, we may easily define the left and right portions of the tape and the optional current symbol (question marks and dots appearing in the code are implicit parameters that the type checker is able to infer by itself):

```
definition left :=λsig.λt:tape sig.match t with
 [ niltape ⇒ [] | leftof _ _ ⇒ [] | rightof s l ⇒ s::l | midtape l _ _ ⇒ l ].

definition right :=λsig.λt:tape sig.match t with
 [ niltape ⇒ [] | leftof s r ⇒ s::r | rightof _ _ ⇒ [] | midtape _ _ r ⇒ r ].

definition current :=λsig.λt:tape sig.match t with
 [ midtape _ c _ ⇒ Some ? c | _ ⇒ None ? ].
```

Note that if $(current\ t) = None$ than either (*left t*) or (*right t*) is empty.

2.2 The Machine

We shall consider machines with three possible moves for the head: L (left) R (right) and N (None).

```
inductive move : Type :=| L : move | R : move | N : move.
```

The machine, parametric over a tape alphabet *sig*, is a record composed of a finite set of *states*, a transition function *trans*, a *start* state, and a set of halting states identified by a boolean function. To encode the alphabet and the states, we exploit the FinSet library of Matita, making extensive use of *unification hints* [7].

```
record TM (sig:FinSet): Type :=
{ states : FinSet;
  trans : states × (option sig) →states × (option (sig × move));
  start : states;
  halt : states →bool}.
```

The transition function takes in input a pair $\langle q, a\rangle$ where q is the current internal state and a is the current symbol of the tape (hence, an optional character); it returns a pair $\langle q, p\rangle$ where p is an *optional* pair $\langle b, m\rangle$ composed of a new character and a move. The rationale is that if we write a new character we will always be allowed to move, also in case the current head symbol was $None$. However, we also want to give the option of not touching the tape (NOP), that is the intended meaning of returning $None$ as output.

Executing p on the tape has the following effect:

```
definition tape_move :=λsig.λt:tape sig.λp:option (sig × move).
  match p with
  [ None ⇒ t
  | Some p1 ⇒
    let ⟨s,m⟩ :=p1 in
    match m with
      [ R ⇒ tape_move_right ? (left ? t) s (right ? t)
      | L ⇒ tape_move_left ? (left ? t) s (right ? t)
      | N ⇒ midtape ? (left ? t) s (right ? t) ] ].
```

where

```
definition tape_move_left :=λsig:FinSet.λlt: list sig.λc:sig.λrt: list sig.
  match lt with
  [ nil ⇒ leftof sig c rt
  | cons c0 lt0 ⇒ midtape sig lt0 c0 (c::rt) ].

definition tape_move_right :=λsig:FinSet.λlt: list sig.λc:sig.λrt: list sig.
  match rt with
  [ nil ⇒ rightof sig c lt
  | cons c0 rt0 ⇒ midtape sig (c::lt) c0 rt0 ].
```

A *configuration* relative to a given set of *states* and an alphabet *sig* is a record composed of a current internal state *cstate* and a *sig* tape.

```
record config (sig,states:FinSet): Type :=
{ cstate : states;
  ctape: tape sig}.
```

A transition *step* between two configurations is defined as follows:

```
definition step :=λsig.λM:TM sig.λc:config sig (states sig M).
  let current_char :=current ? (ctape ?? c) in
  let ⟨news,mv⟩ :=trans sig M ⟨cstate ?? c,current_char⟩ in
  mk_config ?? news (tape_move sig (ctape ?? c) mv).
```

2.3 Computations

A computation is an iteration of the step function until a final internal state is met. In Matita, we may only define total functions, hence we provide an upper bound to the number of iterations, and return an optional configuration depending on the fact that the halting condition has been reached or not.

```
let rec loop (A:Type) n (f:A→ A) p a on n :=
  match n with
  [ O ⇒ None ?
  | S m ⇒ if p a then (Some ? a) else loop A m f p (f a) ].
```

The transformation between configurations induced by a Turing machine M is hence:

```
definition loopM :=λsig,M,i,inc.
  loop ? i (step sig M) (λc.halt sig M (cstate ?? c)) inc.
```

The usual notion of computation for Turing Machines is defined according to given input and output functions, providing the initial tape encoding and the final read-back function. As we know from Kleene's normal form, the output function is particularly important: the point is that our notion of Turing Machine is monotonically increasing w.r.t. tape consumption, with the consequence that the transformation relation between configurations is decidable. However, input and output functions are extremely annoying when composing machines and we would like to get rid of them as far as possible.

Our solution is to define the semantics of a Turing Machine by means of a relation between the input tape and the final tape (possibly embedding the input and output functions): in particular, we say that a machine M *realizes* a relation R between tapes ($M \models R$), if for all t_1 and t_2 there exists a computation leading from $\langle q_o, t_1\rangle$, to $\langle q_f, t_2\rangle$ and $t_1\ R\ t_2$, where q_0 is the initial state and q_f is some halting state of M.

```
definition initc :=λsig.λM:TM sig.λt.
  mk_config sig (states sig M) (start sig M) t.

definition Realize :=λsig.λM:TM sig.λR:relation (tape sig).
∀t.∃i.∃outc.
  loopM sig M i (initc sig M t) = Some ? outc ∧ R t (ctape ?? outc).
```

It is natural to wonder why we use relations on tapes, and not on configurations. The point is that different machines may easily *share* tapes, but they can hardly share their internal states. Working with configurations would force us to an input/output recoding between different machines that is precisely what we meant to avoid.

Our notion of realizability implies termination. It is natural to define a weaker notion (weak realizability, denoted $M|\models R$), asking that $t_1\ R\ t_2$ *provided* there is a computation between t_1 and t_2. It is easy to prove that termination together with weak realizability imply realizability (we shall use the notation $M \downarrow t$ to express the fact that M terminates on input tape t).

```
definition WRealize :=λsig.λM:TM sig.λR:relation (tape sig).
∀t,i,outc.
  loopM sig M i (initc sig M t) = Some ? outc → R t (ctape ?? outc).

definition Terminate :=λsig.λM:TM sig.λt. ∃i,outc.
  loopM sig M i (initc sig M t) = Some ? outc.

lemma WRealize_to_Realize : ∀sig.∀M: TM sig.∀R.
  (∀t.M ↓ t) → M ||= R → M |= R.
```

2.4 A canonical relation

For every machine M we may define a canonical relation, that is the smallest relation weakly realized by M

```
definition R_TM :=λsig.λM:TM sig.λq.λt1,t2.
∃i,outc. loopM ? M i (mk_config ?? q t1) = Some ? outc ∧ t2 = ctape ?? outc.

lemma Wrealize_R_TM : ∀sig,M.
  M ||= R_TM sig M (start sig M).

lemma R_TM_to_R: ∀sig,M,R. ∀t1,t2.
  M ||= R → R_TM ? M (start sig M) t1 t2 → R t1 t2.
```

2.5 The Nop Machine

As a first, simple example, we define a Turing machine performing no operation (we shall also use it in the sequel to force, by sequential composition, the existence of a unique final state).

The machine has a single state that is both initial and final; the transition function is irrelevant, since it will never be executed.

The semantic relation *R_nop* characterizing the machine is just the identity and the proof that the machine realizes it is entirely straightforward.

in this case, states are defined as $initN\ 1$, that is the interval of natural numbers less than 1. This is actually a sigma type containing a natural number m and an (irrelevant) proof that it is smaller than n.

```
definition nop_states :=initN 1.
definition start_nop : initN 1 :=mk_Sig ?? 0 (le_n ... 1).

definition nop :=λalpha:FinSet.
  mk_TM alpha nop_states
  (λp.let ⟨q,a⟩ :=p in ⟨q,None ?⟩)
  start_nop (λ_.true).
```

```
definition R_nop :=λalpha.λt1,t2:tape alpha.t2 = t1.

lemma sem_nop : ∀alpha.nop alpha ⊨ R_nop alpha.
```

3 Composing Machines

Turing Machines are usually reputed to suffer for a lack of compositionality. Our semantic approach, however, allows us to compose them in relatively easy ways. This will give us the opportunity to reason at a higher level of abstraction, rapidly forgetting their low level architecture.

3.1 Sequential composition

The sequential composition $M_1 \cdot M_2$ of two Turing Machines M_1 and M_2 is a new machine having as states the disjoint union of the states of M_1 and M_2. The initial state is the (injection of the) initial state of M_1, and similarly the halting condition is inherited from M_2; the transition function is essentially the disjoint sum of the transition functions of M_1 and M_2, plus a transition leading from the final states of M_1 to the (old) initial state of M_2 (here it is useful to have the possibility of not moving the tape).

```
definition seq_trans :=λsig. λM1,M2 : TM sig.
λp. let ⟨s,a⟩ :=p in
  match s with
  [ inl s1 ⇒
      if halt sig M1 s1 then ⟨inr ... (start sig M2), None ?⟩
      else let ⟨news1,m⟩ :=trans sig M1 ⟨s1,a⟩ in ⟨inl ... news1,m⟩
  | inr s2 ⇒
      let ⟨news2,m⟩ :=trans sig M2 ⟨s2,a⟩ in ⟨inr ... news2,m⟩
  ].

definition seq :=λsig. λM1,M2 : TM sig.
  mk_TM sig
    (FinSum (states sig M1) (states sig M2))
    (seq_trans sig M1 M2)
    (inl ... (start sig M1))
    (λs.match s with [inl _ ⇒ false |inr s2 ⇒ halt sig M2 s2]).
```

If $M_1 \models R_1$ and $M_2 \models R_2$ then $M_1 \cdot M_2 \models R_1 \circ R_2$, that is a very elegant way to express the semantics of sequential composition. The proof of this fact, however, is not as straightforward as one could expect. The point is that M_1 works with its own internal states, and we should "lift" its computation to the states of the sequential machine.

To have an idea of the kind of results we need, here is one of the the key lemmas:

```
lemma loop_lift : ∀A,B,k, lift ,f,g,h, hlift ,c,c1.
  (∀x. hlift  ( lift  x) = h x) →
  (∀x.h x = false → lift  (f  x) = g ( lift  x)) →
  loop A k f h c = Some ? c1 →
  loop B k g  hlift  ( lift  c) = Some ? (lift  ... c1).
```

It says that the result of iterating a function g starting from a lifted configuration *lift* c is the same (up to lifting) as iterating a function f from c provided that

1. a base configuration is halting if and only if its lifted counterpart is halting as well;
2. f and g commute w.r.t. lifting on every non-halting configuration.

3.2 If then else

The next machine we define is an if-then-else composition of three machines M_1, M_2 and M_3 respectively implementing a boolean test, and the two conditional branches. One typical problem of working with single tape machines is the storage of intermediate results: using the tape is particularly annoying, since it requires moving the whole tape back and forward to avoid overwriting relevant information. Since in the case of the if-then-else the result of the test is just a boolean, it makes sense to store it in a state of the machine; in particular we expect to end up in a distinguished final state *qacc* if the test is successful, and in a different state otherwise. This special state *qacc* must be explicitly mentioned when composing the machines. The definition of the if-then-else machine is then straightforward: the states of the new machine are the disjoint union of the states of the three composing machines; the initial state is the initial state of M_1; the final states are the final states of M_2 and M_3; the transition function is the union of the transition functions of the composing machines, where we add new transitions leading from *qacc* to the initial state of M_2 and from all other final states of M_1 to the initial state of M_2.

```
definition if_trans :=λsig.  λM1,M2,M3:TM sig. λq:states sig M1.λp.
let ⟨s,a⟩ :=p in
  match s with
  [ inl  s1  ⇒
      if  halt  sig  M1 s1 then
        if  s1==q then ⟨inr ... (inl ... (start sig M2)), None ?⟩
        else ⟨inr  ... (inr ... (start sig M3)), None ?⟩
      else let  ⟨news1,m⟩ :=trans sig M1 ⟨s1,a⟩ in
       ⟨inl  ... news1,m⟩
  |  inr  s'  ⇒
      match s' with
      [  inl  s2  ⇒ let ⟨news2,m⟩ :=trans sig M2 ⟨s2,a⟩ in
         ⟨inr  ... (inl ... news2),m⟩
      |  inr  s3  ⇒ let ⟨news3,m⟩ :=trans sig M3 ⟨s3,a⟩ in
         ⟨inr  ... (inr ... news3),m⟩ ] ].
```

```
definition ifTM :=λsig. λcondM,thenM,elseM:TM sig.λqacc: states sig condM.
  mk_TM sig
    (FinSum (states sig condM) (FinSum (states sig thenM) (states sig elseM)))
    ( if_trans  sig  condM thenM elseM qacc)
    (inl  ... (start sig  condM))
    (λs.match s with
      [  inl  _  ⇒ false
      |  inr  s'  ⇒ match s' with
        [  inl  s2  ⇒ halt sig thenM s2
        |  inr  s3  ⇒ halt sig elseM s3 ]  ]  ).
```

Our realizability semantics is defined on tapes, and not configurations. In order to observe the accepting state we need to define a suitable variant that we call *conditional realizability*, denoted by $M \models [q : R_1, R_2]$. The idea is that M realizes R_1 if it terminates the computation on q, and R_2 otherwise.

```
definition accRealize :=λsig.λM:TM sig.λacc:states sig M.λRtrue,Rfalse.
∀t.∃i.∃outc.
  loopM sig M i (initc  sig  M t) = Some ? outc ∧
    (cstate  ?? outc = acc → Rtrue t (ctape ?? outc)) ∧
    (cstate  ?? outc ≠  acc → Rfalse t  (ctape ?? outc)).
```

The semantics of the if-then-else machine can be now elegantly expressed in the following way:

```
lemma sem_if: ∀sig.∀M1,M2,M3:TM sig.∀Rtrue,Rfalse,R2,R3,acc.
  M1 ⊨ [acc:  Rtrue,Rfalse]  → M2 ⊨ R2 → M3 ⊨ R3 →
    ifTM sig M1 M2 M3 acc ⊨ (Rtrue ∘R2) ∪(Rfalse ∘ R3).
```

It is also possible to state the semantics in a slightly stronger form: in fact, we know that if the test is successful we shall end up in a final state of M_2 and otherwise in a final state of M_3. If M_2 has a single final state, we may express the semantics by a conditional realizability over this state. As we already observed, a simple way to force a machine to have a unique final state is to sequentially compose it with the nop machine. Then, it is possible to prove the following result (the conditional state is a suitable injection of the unique state of the nop machine):

```
lemma acc_sem_if: ∀sig,M1,M2,M3,Rtrue,Rfalse,R2,R3,acc.
  M1 ⊨ [acc:  Rtrue,  Rfalse]   → M2 ⊨ R2 → M3 ⊨ R3 →
  ifTM sig M1 (single_finalTM ... M2) M3 acc ⊨
      [inr  ... (inl ... (inr ... start_nop)): Rtrue ∘R2, Rfalse ∘ R3].
```

3.3 While

The last machine we are interested in, implements a while-loop over a body machine M. Its definition is really simple, since we have just to add to M a single transition leading from a distinguished final state q back to the initial state.

```
definition while_trans :=λsig. λM : TM sig. λq:states sig M. λp.
  let ⟨s,a⟩ :=p in
  if s == q then ⟨start ? M, None ?⟩
  else trans ? M p.

definition whileTM :=λsig. λM : TM sig. λqacc: states ? M.
  mk_TM sig
    (states ? M)
    (while_trans sig M qacc)
    (start sig M)
    (λs.halt sig M s ∧ ¬ s==qacc).
```

More interesting is the way we can express the semantics of the while machine: provided that $M \models [q : R_1, R_2]$, the while machine (relative to q) weakly realizes $R_1^* \circ R_2$:

```
theorem sem_while: ∀sig,M,qacc,Rtrue,Rfalse.
  halt sig M qacc = true →
  M ⊨ [qacc: Rtrue,Rfalse] →
    whileTM sig M qacc ||⊨ (star ? Rtrue) ∘Rfalse.
```

In this case, the use of weak realizability is essential, since we are not guaranteed to exit the while loop, and the computation can actually diverge. Interestingly, we can reduce the termination of the while machine to the well foundedness of $Rtrue$:

```
theorem terminate_while: ∀sig,M,qacc,Rtrue,Rfalse,t.
  halt sig M qacc = true →
  M ⊨ [qacc: Rtrue,Rfalse] →
  WF ? (inv ... Rtrue) t → whileTM sig M qacc ↓t.
```

4 Basic Machines

A major mistake we made when we started implementing the universal machine consisted in modelling relatively complex behaviors by directly writing a corresponding Turing Machine. While writing the code is usually not very complex, proving its correctness is often a nightmare, due to the complexity of specifying and reasoning about internal states of the machines and all intermediate configurations. A much better approach consists in specifying a small set of basic machines, and define all other machines by means of the compositional constructs of the previous section. In this way, we may immediately forget about Turing Machines' internals, since the behavior of the whole program only depends on the behavior of its components.

A very small set of primitive programs turned out to be sufficient for our purposes (most of them are actually families of machines, parametrized over some input arguments)

write c write the character c on the tape at the current head position
move_r move the head one step to the right
move_l move the head one step to the left
test_char f perform a boolean test f on the current character and enter state tc_true or tc_false according to the result of the test
swap_r swap the current character with its right neighbor (if any)
swap_l swap the current character with its left neighbor (if any)

The specification of these machines is straightforward. Let us have a glance at the *swap_r* machine. In order to swap characters we need an auxiliary memory cell; since tape characters are finite, we may use an internal state (register) of the machine to this purpose. The machine will sequentially enter in the following four states:

- swap0: read the current symbol, save it in a register and move right
- swap1: swap the current symbol with the register content, and move back to the left
- swap2: write the register content at the current position
- swap3: stop

Here is the machine implementation:

```
definition swap_r :=
 λalpha:FinSet.λfoo:alpha.
 mk_TM alpha (swap_states alpha)
 (λp.let ⟨q,a⟩ :=p in
  let ⟨q',b⟩ :=q in
  let q' :=\fst q' in (* extract the witness *)
  match a with
  [ None ⇒ ⟨⟨swap3,foo⟩,None ?⟩ (* if tape is empty then stop *)
  | Some a' ⇒
  match q' with
  [ O ⇒ (* q0 *) ⟨⟨swap1,a'⟩,Some ? ⟨a',R⟩⟩ (* save in register and move R *)
  | S q' ⇒ match q' with
    [ O ⇒ (* q1 *) ⟨⟨swap2,a'⟩,Some ? ⟨b,L⟩⟩ (* swap with register and move L *)
    | S q' ⇒ match q' with
      [ O ⇒ (* q2 *) ⟨⟨swap3,foo⟩,Some ? ⟨b,N⟩⟩ (* copy from register and stay *)
      | S q' ⇒ (* q3 *) ⟨⟨swap3,foo⟩,None ?⟩ (* final state *)
      ] ] ] ] )
 ⟨swap0,foo⟩
 (λq.\fst q == swap3).
```

and this is its specification.

```
definition Rswap_r :=λalpha,t1,t2.
  ∀a,b,ls,rs. t1 = midtape alpha ls b (a::rs) →t2 = midtape alpha ls a (b::rs).
```

It is possibly worth to remark that an advantage of using relations is the possibility of under-specifying the behavior of the program, restricting the attention

to what we expect to be the structure of the input (e.g., in the previous case, the fact of receiving a mid-tape as the input tape).

The proof that *swap_r* realizes its specification is by cases on the structure of the tape: three cases are vacuous; the case when the tape is actually a mid-tape is essentially solved by direct computation.

4.1 Composing machines

Let us see an example of how we can use the previous bricks to build more complex functions. When working with Turing Machines, moving characters around the tape is a very frequent and essential operation. In particular, we would like to write a program that moves a character to the left until we reach a special character taken as a parameter (*move_char_l*). A step of the machine essentially consists of a swap operation, but guarded by a conditional test; then we shall simply wrap a while machine around this step.

```
definition mcl_step :=λalpha:FinSet.λsep:alpha.
  ifTM alpha (test_char ? (λc.¬ c==sep))
     (single_finalTM ... (swap_r alpha sep · move_l ?)) (nop ?) tc_true.

definition Rmcl_step_true :=λalpha,sep,t1,t2.
   ∀a,b, ls , rs .
    t1 = midtape alpha ls b (a:: rs) →
    b ≠ sep ∧ t2 = mk_tape alpha (tail ? ls) (option_hd ? ls) (a:: b:: rs).

definition Rmcl_step_false :=λalpha,sep,t1,t2.
     right ? t1 ≠ [] → current alpha t1 ≠ None alpha →
       current alpha t1 = Some alpha sep ∧ t2 = t1.

definition mcls_acc: ∀alpha:FinSet.∀sep:alpha.states ? (mcl_step alpha sep)
 :=λalpha,sep.inr ... (inl ... (inr ... start_nop)).

lemma sem_mcl_step :
  ∀alpha,sep.
  mcl_step alpha sep ⊨
    [mcls_acc alpha sep: Rmcl_step_true alpha sep, Rmcl_step_false alpha sep]
```

Here is the full *move_char_l* program:

```
definition move_char_l :=λalpha,sep.
  whileTM alpha (mcl_step alpha sep) (mlcs_acc alpha sep).

definition R_move_char_l :=λalpha,sep,t1,t2.
  ∀b,a, ls , rs . t1 = midtape alpha ls b (a:: rs) →
   (b = sep → t2 = t1) ∧
    (∀ls1 , ls2 . ls = ls1@sep:: ls2 →
     b ≠ sep → memb ? sep ls1 = false →
      t2 = midtape alpha ls2 sep (a:: reverse ? ls1@b:: rs )).
```

```
lemma sem_move_char_l : ∀alpha,sep.
  WRealize alpha (move_char_l alpha sep) (R_move_char_l alpha sep).
```

In a very similar way, we may define two machines *move_left_to* and *move_right_to* that move the head left or right until they meet a character that satisfies a given condition.

5 Normal Turing Machines

A normal Turing machine is just an ordinary machine where:

1. the tape alphabet is $\{0, 1\}$;
2. the finite states are supposed to be an initial interval of the natural numbers.

By convention, we assume the starting state is 0.

```
record normalTM : Type :=
{ no_states : nat;
  pos_no_states : (0 < no_states);
  ntrans : (initN no_states)×Option bool → (initN no_states)×Option (bool×Move);
  nhalt : initN no_states →bool}.
```

We may easily define a transformation from a normal TM into a traditional Machine; declaring it as a coercion we allow the type system to freely convert the former into the latter:

```
definition normalTM_to_TM :=λM:normalTM.
  mk_TM FinBool (initN (no_states M))
   (ntrans M) (mk_Sig ?? 0 (pos_no_states M)) (nhalt M).

coercion normalTM_to_TM.
```

A normal configuration is a configuration for a normal machine: it only depends on the number of states of the normal Machine:

```
definition nconfig :=λn. config FinBool (initN n).
```

5.1 Tuples

By general results on FinSets (the Matita library about finite sets) we know that every function f between two finite sets A and B can be described by means of a finite graph of pairs $\langle a, fa\rangle$. Hence, the transition function of a normal Turing machine can be described by a finite set of tuples $\langle\langle i, c\rangle, \langle j, action\rangle\rangle$ of the following type:

$$(initN\ n \times option\ bool) \times (initN\ n \times option\ bool \times move)$$

Unfortunately, this description is not suitable for a Universal Machine, since such a machine must work with a fixed set of states, while the size on n is unknown. Hence, we must pass from natural numbers to a representation for them on a finitary, e.g. binary, alphabet. In general, we shall associate to a pair $\langle\langle i, c\rangle, \langle j, action\rangle\rangle$ a tuple with the following syntactical structure

$$|w_i x, w_j y, z$$

where

1. "|" and "," are special characters used as delimiters;
2. w_i and w_j are list of booleans representing the states i and j;
3. x is special symbol *null* if $c = None$ and is the boolean a if $c = Some\ a$
4. y and z are both *null* if $action = None$, and are respectively equal to b and m' if $action = Some(b, m)$
5. finally, $m' = 0$ if $m = L$, $m' = 1$ if $m = R$ and $m' = null$ if $m = N$

As a minor, additional complication, we shall suppose that every character is decorated by an additional bit, normally set to false, to be used as a marker.

```
definition mk_tuple :=λqin,cin,qout,cout,mv.
 ⟨bar, false⟩ :: qin @ cin :: ⟨comma,false⟩ :: qout @ cout :: ⟨comma,false⟩ :: [mv].
```

The actual encoding of states is not very important, and we shall skip it: the only relevant points are that (a) it is convenient to assume that all states (and hence all tuples for a given machine) have a fixed, uniform length; (b) the first bit of the representation of the state tells us if the state is final or not.

5.2 The table of tuples

The list of all tuples, concatenated together, provides the low level description of the normal Turing Machine to be interpreted by the Universal Machine: we call it a *table*.

The main lemma relating a table to the corresponding transition function is the following one, stating that for a pair $\langle s, t\rangle$ belonging to the graph of *trans*, and supposing that l is its encoding, then l occurs as a sublist (can be matched) inside the table associated with *trans*.

```
lemma trans_to_match:
  ∀n.∀h.∀trans: trans_source n →trans_target n.
  ∀inp,outp,qin,cin,qout,cout,mv. trans inp = outp →
  tuple_encoding n h ⟨inp,outp⟩ = mk_tuple qin cin qout cout mv →
  match_in_table (S n) qin cin qout cout mv
    (flatten ? ( tuples_list n h (graph_enum ?? trans))).
```

5.3 The use of marks

We shall use a special alphabet where every character can be marked with an additional boolean. Marks are typically used in pairs and are meant to identify (and recall) a source and a target position where some joint operation must be performed: typically, a comparison or a copy between strings. The main generic operations involving marks are the following:

mark mark the current cell
clear_mark clear the mark (if any) from the current cell
adv_mark_r shift the mark one position to the right
adv_mark_l shift the mark one position to the left
adv_both_marks shift the marks at the right and left of the head one position to the right
match_and_advance f if the current character satisfies the boolean test f then advance both marks and otherwise remove them
adv_to_mark_r move the head to the next mark on the right
adv_to_mark_l move the head to the next mark on the left

5.4 String comparison

Apart from markings, there is an additional small problem in comparing and copying strings. The natural idea would be to store the character to be compared/copied into a register (i.e. as part of the state); unfortunately, our semantics is not state-aware. The alternative solution we have exploited is to have a family of machines, each specialized on a given character. So, comparing a character will consist of testing a character and calling the suitable machine in charge of checking/writing that particular character at the target position. This behavior is summarized in the following functions. The *comp_step_subcase* takes as input a character c, and a continuation machine $elseM$ and compares the current character with c; if the test succeeds it moves to the next mark to the right, repeats the comparison, and if successful advances both marks; if the current character is not c, it passes the control to $elseM$.

```
definition comp_step_subcase :=λalpha,c,elseM.
  ifTM ? (test_char ? (λx.x == c))
    (move_r ... · adv_to_mark_r ? (is_marked alpha) · match_and_adv ? (λx.x == c))
    elseM tc_true.
```

A step of the *compare* machine consists in using the previous function to build a chain of specialized testing functions on all characters we are interested in (in this case, *true*, *false*, or *null*), each one passing control to the next one in cascade:

```
definition comp_step :=
  ifTM ? (test_char ? (is_marked ?))
  (single_finalTM ... (comp_step_subcase FSUnialpha ⟨bit false,true⟩
    (comp_step_subcase FSUnialpha ⟨bit true,true⟩
      (comp_step_subcase FSUnialpha ⟨null,true⟩
        (clear_mark ... )))))
  (nop ?)
  tc_true.
```

String comparison is then simply a while over *comp_step*

```
definition compare :=
  whileTM ? comp_step (inr ... (inl ... (inr ... start_nop))).
```

6 The Universal Machine

Working with a single tape, the most efficient way to simulate a given machine M is by keeping its code always close to the head of the tape, in such a way that the cost of fetching the next move is independent of the current size of the tape and only bounded by the dimension of M. The drawback is that simulating a tape move can require to shift the whole code of M; assuming however that this is fixed, we have no further complexity slow-down in the interpretation. The Universal Machine is hence *fair* in the sense of [3].

Our universal machine will work with an alphabet comprising booleans and four additional symbols: "*null*", "#" (grid), "|" (bar) and "," (comma). In addition, in order to compare cells and to move their content around, it is convenient to assume the possibility of marking individual cells: so our tape symbols will actually be pairs of an alphabet symbol and a boolean mark (usually set to false).

The universal machine must be ready to simulate machines with an arbitrary number of states. This means that the current state of the simulated machine cannot be kept in a register (state) of the universal machine, but must be memorized on the tape. We keep it together with the current symbol of the simulated tape

The general structure of the tape is the following:

$$\alpha\# \overset{\Downarrow}{q_{i0}} \dots q_{in} c\# table\#\beta$$

where α, β and c are respectively the left tape, right tape, and current character of the simulated machine. If there is no current character (i.e. the tape is empty or we are in a left or right overflow) then c is the special "*null*" character. The string $w_i = q_{i0} \dots q_{in}$ is the encoding of the current state q_i of M, and *table* is the set of tuples encoding the transition function of M, according to the definition of the previous section. In a well formed configuration we always have three occurrences of #: a leftmost, a middle and rightmost one; they are

basic milestones to help the machine locating the information on the tape. At each iteration of a single step of M the universal machine will start with its head (depicted with $\Downarrow$ in the above representation) on the first symbol q_{i0} of the state.

Each step is simulated by performing two basic operations: fetching in the table a tuple matching $w_i c$ (*match_tuple*), and executing the corresponding action (*exec_action*). The *exec_action* function is also responsible for updating $w_i c$ according to the new state-symbol pair $w_j d$ provided by the matched tuple.

If matching succeeds, *match_tuple* is supposed to terminate in the following configuration, with the head on the middle #

$$\alpha \# w_i c \overset{\Downarrow}{\#} \underbrace{\ldots |w_i c \overset{*}{,} w_j d, m| \ldots}_{table} \# \beta$$

where moreover the comma preceding the action to be executed will be marked (marking will be depicted with a $*$ on top of the character). If matching fails, the head will be on the # at the end of table (marked to discriminate easily this case from the former one):

$$\alpha \# w_i c \# table \overset{\Downarrow}{\underset{*}{\#}} \beta$$

The body of the universal machine is hence the following *uni_step* function, where *tc_true* is the accepting state of the *test_char* machine (in the next section we shall dwell into the details of the *match_tuple* and *exec_action* functions).

```
definition uni_step :=
 ifTM ? (test_char STape (λc.\fst c == bit false))
    (single_finalTM ?
      (init_match · match_tuple ·
        (ifTM ? (test_char ? (λc.¬ is_marked ? c))
          (exec_action · move_r ...)
          (nop ?) tc_true )))
    (nop ?) tc_true .
```

At the end of *exec_action* we must perform a small step to the right to reenter the expected initial configuration of *uni_step*.

The universal machine is simply a while over *uni_step*:

```
definition universalTM :=whileTM ? uni_step us_acc.
```

The main semantic properties of *uni_step* and *universalTM* will be discussed in Section 9.

7 Matching

Comparing strings on a single tape machine requires moving back and forth between the two stings, suitably marking the corresponding positions on the

tape. The following *initialize_match* function initializes marks, adding a mark at the beginning of the source string (the character following the leftmost #, where the current *state*, *character* pair begins), and another one at the beginning of the table (the character following the middle #):

```
definition init_match :=
  mark ? · adv_to_mark_r ? (λc:STape.is_grid (\fst c)) · move_r ? ·
    move_r ? · mark ? · move_l ? · adv_to_mark_l ? (is_marked ?).
```

The *match_tuple* machine scrolls through the tuples in the transition table until one matching the source string is found. It just repeats, in a while loop, the operation of trying to match a single tuple discussed in the next section:

```
definition match_tuple :=
  whileTM ? match_tuple_step (inr ... (inl ... (inr ... start_nop))).
```

7.1 *match_tuple_step*

The *match_tuple_step* starts checking the halting condition, that is when we have reached a (rightmost) #. If this is not the case, we execute the "then" branch, where we compare the two strings starting from the marked characters. If the two strings are equal, we mark the comma following the matched string in the table and then we stop on the middle #; otherwise, we mark the next tuple (if any) and reinitialize the mark at the beginning of the current state-character pair. If there is no next tuple, we stop on the rightmost grid after marking it.

If on the contrary the *match_tuple_step* is executed when the current character is a #, we execute the "else" branch, which does nothing.

```
definition match_tuple_step :=
  ifTM ? (test_char ? (λc:STape.¬ is_grid (\fst c)))
   (single_finalTM ?
     (compare ·
      (ifTM ? (test_char ? (λc:STape.is_grid (\fst c)))
        (nop ?)
        (mark_next_tuple ·
           (ifTM ? (test_char ? (λc:STape.is_grid (\fst c)))
             (mark ?) (move_l ? · init_current ) tc_true )) tc_true )))
    (nop ?) tc_true .
```

The *match_tuple_step* is iterated until we end up in the "else" branch, meaning the head is reading a #. The calling machine can distinguish whether we ended up in a failure or success state depending on whether the # is marked or not.

8 Action Execution

Executing an action can be decomposed in two simpler operations, which can be executed sequentially: updating the current state and the character under the

(simulated) tape head (*copy*), and moving the (simulated) tape (*move_tape*). Similarly to matching, copying is done one character at a time, and requires a suitable marking of the tape (and a suitable initialization *init_copy*). As we shall see, the *copy* machine will end up clearing all marks, halting with the head on the comma preceding the tape move. Since *tape_move* expects to start with the head on the move, we must move the head one step to the right before calling it.

```
definition exec_action :=
  init_copy  ·  copy  ·  move_r ... · move_tape.
```

8.1 *init_copy*

The *init_copy* machine initializes the tape marking the positions corresponding to the the cell to be copied and its destination (with the head ending up on the former). In our case, the destination is the position on the right of the leftmost #, while the source is the action following the comma in the tuple that has been matched in the table (that is the position to the right of the currently marked cell). In graphical terms, the *init_copy* machine transforms a tape of the form

$$\alpha\#q_{i0}\dots q_{in}c\#\underbrace{\dots|w_k a \overset{\Downarrow}{\overset{*}{,}} q_{j0}\dots q_{jn}d,m|\dots}_{table}\#\beta$$

into

$$\alpha\#\overset{*}{q_{i0}}\dots q_{in}c\#\underbrace{\dots|w_k a,\overset{\Downarrow}{\overset{*}{q_{j0}}}\dots q_{jn}d,m|\dots}_{table}\#\beta$$

This is the corresponding code:

```
definition init_copy :=
  init_current_on_match  ·  move_r ?  ·
    adv_to_mark_r ? (is_marked ?)  ·  adv_mark_r ?.
```

where

```
definition init_current_on_match :=
  move_l ?  ·  adv_to_mark_l ? (λc:STape.is_grid  (\fst  c))  ·  move_r ?  ·  mark ?.
```

8.2 *copy*

The copy machine copies the portion of the tape starting on the left mark and ending with a comma to a portion of the tape of the same length starting on the right mark. The machine is implemented as a while machine whose body copies

one bit at a time, and advances the marks. In our case, this will allow us to pass from a configuration of the kind

$$\alpha\#\ \overset{*}{q_{i0}}\ldots q_{in}c\#\underbrace{\ldots|w_ka,\overset{\Downarrow}{\overset{*}{q_{j0}}}\ldots q_{jn}d,m|\ldots}_{table}\#\beta$$

to a configuration like

$$\alpha\#q_{j0}\ldots q_{jn}d\#\underbrace{\ldots|w_ka,q_{j0}\ldots q_{jn}d\overset{\Downarrow}{,}m|\ldots}_{table}\#\beta$$

As a special case, d can be a *null* rather than a bit: this identifies those actions that do not write a character to the tape. The *copy* machine acts accordingly, ignoring *null*s and leaving c untouched.

Note that the copy machine will remove all marks before exiting.

8.3 *move_tape*

Finally, the *move_tape* machine mimics the move action on the simulated tape. This is a complex operation, since we must skip the code of the simulated machine and its state. The main function just tests the character encoding the move action and calls three more elementary functions: *move_tape_r*, *move_tape_l*, and *no_move*:

```
definition move_tape :=
  ifTM ? (test_char ? (λc:STape.c == ⟨bit false, false⟩))
    (adv_to_mark_r ? (λc:STape.is_grid (\fst c)) · move_tape_l)
    (ifTM ? (test_char ? (λc:STape.c == ⟨bit true,false⟩))
       (adv_to_mark_r ? (λc:STape.is_grid (\fst c)) · move_tape_r)
       (no_move ?) tc_true) tc_true.
```

The *no_move* machine is pretty simple since it is merely responsible for resetting the head of tape at the expected output position, that is on the leftmost #:

```
definition no_move :=
  adv_to_mark_l ? (λc:STape.is_grid (\fst c)) ·
    move_l ... · adv_to_mark_l ? (λc:STape.is_grid (\fst c)))
```

The other two functions are pretty similar and we shall only discuss the first one.

8.4 *move_tape_r*

The move tape right is conceptually composed of three sub-functions, executed sequentially: a *fetch_r* function, that advances the head to the first character of the simulated right tape (that is, the first character after the rightmost #), and initializes it to *null* if the tape is empty; a *set_new_current_r* function that moves

it to the "current" position, that is at the position at the left of the middle #; and finally a *move_old_current_r*, that moves the old "current" value (which is now just at the left of the tape head), as first symbol of the left tape (that is, just after the the leftmost #). The last two functions are in fact very similar: they have just to move a character after the first # at their left (*move_after_left_grid*)

This is the evolution of the tape, supposing the right tape is not empty:

$$\begin{array}{ll}
\alpha\#w_jd\#table\overset{\Downarrow}{\#}b\beta & fetch_r \\
\alpha\#w_jd\#table\overset{\Downarrow}{b}\#\beta & move_after_left_grid \\
\alpha\#w_jd\overset{\Downarrow}{b}\#table\#\beta & move_l \\
\alpha\#w_j\overset{\Downarrow}{d}b\#table\#\beta & move_after_left_grid \\
\alpha\overset{\Downarrow}{d}\#w_jb\#table\#\beta & move_r \\
\alpha d\overset{\Downarrow}{\#}w_jb\#table\#\beta &
\end{array}$$

This is the code for the above machines:

```
definition fetch_r :=
  move_r ... · init_cell · move_l ... · swap_r STape ⟨grid,false⟩.

definition move_after_left_grid :=
  move_l ... · move_char_l STape ⟨grid,false⟩ · swap_r STape ⟨grid,false⟩.

definition move_tape_r :=
  fetch_r · move_after_left_grid · move_l ... · move_after_left_grid · move_r ....
```

init_cell is an atomic machine defined in the obvious way.

9 Main Results

Given a configuration for a normal machine M, the following function builds the corresponding "low level" representation, that is the actual tape manipulated by the Universal Machine:

```
definition low_config: ∀M:normalTM.nconfig (no_states M) → tape STape :=
λM:normalTM.λc.
  let n :=no_states M in
  let h :=nhalt M in
  let t :=ntrans M in
  let q :=cstate ... c in
  let q_low := m_bits_of_state n h q in
  let current_low :=
    match current ... (ctape ... c) with
    [ None ⇒ null | Some b ⇒ bit b] in
  let low_left :=map ... (λb.⟨bit b,false⟩) (left ... (ctape ... c)) in
  let low_right :=map ... (λb.⟨bit b,false⟩) (right ... (ctape ... c)) in
```

```
  let table :=flatten ? ( tuples_list  n h (graph_enum ?? t)) in
  let right :=
    q_low@⟨current_low,false⟩ :: ⟨grid , false ⟩ :: table@⟨grid, false ⟩ :: low_right  in
  mk_tape STape (⟨grid,false⟩ :: low_left )  (option_hd ... right) (tail  ... right).
```

Similarly, every relation over tapes can be reflected into a corresponding relation on their low-level representations:

```
definition low_R :=λM,qstart,R,t1,t2.
    ∀tape1. t1 = low_config M (mk_config ?? qstart tape1) →
    ∃q,tape2.R tape1 tape2 ∧
    halt ? M q = true ∧ t2 = low_config M (mk_config ?? q tape2).
```

We expect the Universal Machine to be able to simulate on its tape each step of the machine M, and to stop leaving the tape unchanged when M stops. The machine must be able to end up in a special accepting state *us_acc* in the former case, and in a different state in the latter. The input-output relation realized by the machine in the two cases are the following:

```
definition low_step_R_true :=λt1,t2.
  ∀M:normalTM.∀c: nconfig (no_states M).
    t1 = low_config M c →
    halt ? M (cstate ... c) = false ∧ t2 = low_config M (step ? M c).

definition low_step_R_false :=λt1,t2.
  ∀M:normalTM.
  ∀c: nconfig (no_states M).
    t1 = low_config M c → halt ? M (cstate ... c) = true ∧ t1 = t2.

lemma sem_uni_step1:
  uni_step ⊨ [us_acc: low_step_R_true,  low_step_R_false ].
```

For the universal machine we proved that, for any normal machine M, it weakly realizes the low level version of the canonical relation for M

```
theorem sem_universal: ∀M:normalTM. ∀qstart.
  universalTM |⊨ (low_R M qstart (R_TM FinBool M qstart)).
```

From this result it is easy to derive that, for any relation weakly realized by M, the universal machine weakly realizes its low level counterpart.

```
theorem sem_universal2: ∀M:normalTM. ∀R.
  M |⊨ R → universalTM |⊨ (low_R M (start ? M) R).
```

Termination is stated by the following result, whose proof is still in progress.

```
theorem terminate_UTM: ∀M:normalTM.∀t.
  M ↓ t → universalTM ↓(low_config  M (mk_config ?? (start ? M) t)).
```

10 Conclusions

We provided in this paper some preliminary results about formal specification and verification of Turing Machines, up to the definition of a universal machine and the proof of its correctness. The work is organized in 15 files (see Figure 1), for a total of 6743 lines (comprising comments). It has been developed by the two authors during 2.5 months of intense joint work, at the good rate of more than 300 lines per man-week (see [4] for an estimation of the cost of formalization at the current state of the art).

name	dimension	content
mono.ma	475 lines	mono-tape Turing machines
if_machine.ma	335 lines	conditional composition
while_machine	166 lines	while composition
basic_machines.ma	282 lines	basic atomic machines
move_char.ma	310 lines	character copying
alphabet.ma	110 lines	alphabet of the universal machine
marks.ma	901 lines	operations exploiting marks
compare.ma	506 lines	string comparison
copy.ma	579 lines	string copy
normalTM.ma	319 lines	normal Turing machines
tuples.ma	276 lines	normal Turing machines
match_machines.ma	727 lines	machines implementing matching
move_tape.ma	778 lines	machines for moving the simulated tape
uni_step.ma	585 lines	emulation of a high-level step
universal.ma	394 lines	the universal machine
total	6743 lines	

Fig. 1. List of files and their dimension in lines

.

One could possibly wonder what is the actual purpose for performing a similar effort, but the real question is in fact the opposite one: what could be the reason for *not doing* it, since it requires a relatively modest investment in time and resources? The added value of having a complete, executable, and automatically verifiable specification is clear, and it could certainly help to improve confidence (of students, if not of researchers) in a delicate topic that, especially in modern textbooks, is handled in a very superficial way.

The development presented in this paper is still very preliminary, under many respects. In particular, the fact that the universal machine operates with a different alphabet with respect to the machines it simulates is annoying. Of course, any machine can be turned into a normal Turing machine, but this transformation may require a recoding of the alphabet that is not entirely transparent to complexity issues: for example, prefixing every character in a string $x_1 \dots x_n$ with a 0 in order to get the new string $0x_1 \dots 0x_n$ could take, on a single tape

Turing Machine, a time quadratic in the length n of the string (this is precisely the kind of problems that raises a legitimate suspicion on the actual complexity of a *true* interpreter).

Complexity Theory, more than Computability, is indeed the real, final target of our research. Any modern textbook in Complexity Theory (see e.g. [2]) starts with introducing Turing Machines just to claim, immediately after, that the computational model *does not matter*. The natural question we are addressing and that we hope to contribute to clarify is: *what* matters?

The way we plan to attack the problem is by *reversing* the usual deductive practice of deriving theorems from axioms, reconstructing from proofs the basic assumptions underlying the major notions and results of Complexity Theory. The final goal of our Reverse Complexity Program is to obtain a formal, axiomatic treatment of Complexity Theory at a *comfortable* level of abstraction, providing in particular logical characterizations of Complexity Classes that could help to better grasp their essence, identify their distinctive properties, suggest new, possibly non-standard computational models and finally provide new tools for *separating* them.

The axiomatization must obviously be validated with respect to traditional cost models, and in particular w.r.t. Turing Machines that still provide the actual foundation for this discipline. Hence, in conjunction with the "reverse" approach, it is also important to promote a more traditional forward approach, deriving out of concrete models the key ingredients for the study of their complexity aspects. The work in this paper, is meant to be a contribution along this second line of research.

References

1. R. Amadio, Andrea Asperti, Nicholas Ayache, B. Campbell, D. Mulligan, R. Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, and I. Stark. Certified complexity. *Procedia CS*, 7:175–177, 2011.
2. Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge Univ. Press, 2009.
3. Andrea Asperti. The intensional content of rice's theorem. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), January 7-12, 2008, San Francisco, California, USA*, pages 113–119. ACM, 2008.
4. Andrea Asperti and Claudio Sacerdoti Coen. Some considerations on the usability of interactive provers. In *Intelligent Computer Mathematics, 10th International Conference, Paris, France, July 5-10, 2010*, volume 6167 of *Lecture Notes in Computer Science*, pages 147–156. Springer, 2010.
5. Andrea Asperti and Jeremy Avigad (eds). Special issue on interactive theorem proving and the formalisation of mathematics. *Mathematical Structures in Computer Science*, 21(4), 2011.
6. Andrea Asperti and Wilmer Ricciotti. A web interface for matita. In *Proceedings of Intelligent Computer Mathematics (CICM 2012), Bremen, Germany*, volume 7362 of *Lecture Notes in Artificial Intelligence*. Springer, 2012.

7. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.
8. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE-2011), Wroclaw, Poland*, volume 6803 of *LNCS*, 2011.
9. Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Log.*, 9(1), 2007.
10. Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA*, pages 3–15. ACM, 2008.
11. Martin Davis. *Computability and Unsolvability.* Dover Publications, 1985.
12. R. E. Stearns F. C. Hennie. Two-tape simulation of multi tape turing machines. *Journal of ACM*, 13(4):533–546, 1966.
13. Patrick C. Fischer. On formalisms for turing machines. *J. ACM*, 12(4):570–580, 1965.
14. Thomas Hales, Georges Gonthier, John Harrison, and Freek Wiedijk. A Special Issue on Formal Proof. *Notices of the American Mathematical Society*, 55, 2008.
15. J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transaction of the American Mathematical Society*, 117:285–306, 1965.
16. F. C. Hennie. One-tape, off-line turing machine computations. *Information and Control*, 8(6):553–578, 1965.
17. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.
18. Gerwin Klein. Operating system verification – an overview. *Sadhana*, 34(1):27–69, 2009.
19. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA*, pages 42–54, 2006.
20. Michael Norrish. Mechanised computability theory. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.
21. Michael Sipser. *Introduction to the Theory of Computation.* PWS, 1996.
22. Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. of the London Math. Society*, 2(42):230–265, 1936.
23. Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. 1990.
24. Stephanie Weirich and Benjamin Pierce (eds). Special issue on the poplmark challenge. *Journal of Automated Reasoning*, 2011. Published online.
25. Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *Commun. ACM*, 54(12):123–131, 2011.