Mechanising Turing Machines and Computability Theory in Isabelle/HOL

Jian Xu[1], Xingyuan Zhang[1], and Christian Urban[2]

[1] PLA University of Science and Technology, China
[2] King's College London, UK

Abstract. We present a formalised theory of computability in the theorem prover Isabelle/HOL. This theorem prover is based on classical logic which precludes *direct* reasoning about computability: every boolean predicate is either true or false because of the law of excluded middle. The only way to reason about computability in a classical theorem prover is to formalise a concrete model for computation. We formalise Turing machines and relate them to abacus machines and recursive functions. We also formalise a universal Turing machine and Hoare-style reasoning techniques that allow us to reason about Turing machine programs. Our theory can be used to formalise other computability results. We give one example about the computational equivalence of single-sided Turing machines.

1 Introduction

Suppose you want to mechanise a proof for whether a predicate P, say, is decidable or not. Decidability of P usually amounts to showing whether $P \vee \neg P$ holds. But this does *not* work in Isabelle/HOL and other HOL theorem provers, since they are based on classical logic where the law of excluded middle ensures that $P \vee \neg P$ is always provable no matter whether P is constructed by computable means. We hit on this limitation previously when we mechanised the correctness proofs of two algorithms [6,7], but were unable to formalise arguments about decidability.

The only satisfying way out of this problem in a theorem prover based on classical logic is to formalise a theory of computability. Norrish provided such a formalisation for the HOL. He choose the λ-calculus as the starting point for his formalisation of computability theory, because of its "simplicity" [4, Page 297]. Part of his formalisation is a clever infrastructure for reducing λ-terms. He also established the computational equivalence between the λ-calculus and recursive functions. Nevertheless he concluded that it would be appealing to have formalisations for more operational models of computations, such as Turing machines or register machines. One reason is that many proofs in the literature use them. He noted however that [4, Page 310]:

> *"If register machines are unappealing because of their general fiddliness, Turing machines are an even more daunting prospect."*

In this paper we take on this daunting prospect and provide a formalisation of Turing machines, as well as abacus machines (a kind of register machines) and recursive functions. To see the difficulties involved with this work, one has to understand that Turing

machine programs can be completely *unstructured*, behaving similar to Basic programs involving the infamous goto [3]. This precludes in the general case a compositional Hoare-style reasoning about Turing programs. We provide such Hoare-rules for when it *is* possible to reason in a compositional manner (which is fortunately quite often), but also tackle the more complicated case when we translate abacus programs into Turing programs. This reasoning about Turing machine programs is usually completely left out in the informal literature, e.g. [2].

We are not the first who formalised Turing machines: we are aware of the preliminary work by Asperti and Ricciotti [1]. They describe a complete formalisation of Turing machines in the Matita theorem prover, including a universal Turing machine. They report that the informal proofs from which they started are *not* "sufficiently accurate to be directly usable as a guideline for formalization" [1, Page 2]. For our formalisation we follow mainly the proofs from the textbook [2] and found that the description there is quite detailed. Some details are left out however: for example, it is only shown how the universal Turing machine is constructed for Turing machines computing unary functions. We had to figure out a way to generalise this result to n-ary functions. Similarly, when compiling recursive functions to abacus machines, the textbook again only shows how it can be done for 2- and 3-ary functions, but in the formalisation we need arbitrary functions. But the general ideas for how to do this are clear enough in [2].

The main difference between our formalisation and the one by Asperti and Ricciotti is that their universal Turing machine uses a different alphabet than the machines it simulates. They write [1, Page 23]:

> *"In particular, the fact that the universal machine operates with a different alphabet with respect to the machines it simulates is annoying."*

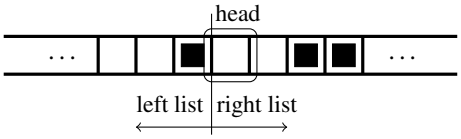
In this paper we follow the approach by Boolos et al [2], which goes back to Post [5], where all Turing machines operate on tapes that contain only *blank* or *occupied* cells. Traditionally the content of a cell can be any character from a finite alphabet. Although computationally equivalent, the more restrictive notion of Turing machines in [2] makes the reasoning more uniform. In addition some proofs *about* Turing machines are simpler. The reason is that one often needs to encode Turing machines—consequently if the Turing machines are simpler, then the coding functions are simpler too. Unfortunately, the restrictiveness also makes it harder to design programs for these Turing machines. In order to construct a universal Turing machine we therefore do not follow [1], instead follow the proof in [2] by translating abacus machines to Turing machines and in turn recursive functions to abacus machines. The universal Turing machine can then be constructed as a recursive function.

Contributions: We formalised in Isabelle/HOL Turing machines following the description of Boolos et al [2] where tapes only have blank or occupied cells. We mechanise the undecidability of the halting problem and prove the correctness of concrete Turing machines that are needed in this proof; such correctness proofs are left out in the informal literature. For reasoning about Turing machine programs we derive Hoare-rules. We also construct the universal Turing machine from [2] by translating recursive functions to abacus machines and abacus machines to Turing machines. Since we have set up in Isabelle/HOL a very general computability model and undecidability result, we

are able to formalise other results: we describe a proof of the computational equivalence of single-sided Turing machines, which is not given in [2], but needed for formalising the undecidability proof of Wang's tiling problem. *citation*

2 Turing Machines

Turing machines can be thought of as having a *head*, "gliding" over a potentially infinite tape. Boolos et al [2] only consider tapes with cells being either blank or occupied, which we represent by a datatype having two constructors, namely *Bk* and *Oc*. One way to represent such tapes is to use a pair of lists, written (l, r), where l stands for the tape on the left-hand side of the head and r for the tape on the right-hand side. We have the convention that the head, abbreviated *hd*, of the right-list is the cell on which the head of the Turing machine currently scannes. This can be pictured as follows:



Note that by using lists each side of the tape is only finite. The potential infinity is achieved by adding an appropriate blank or occupied cell whenever the head goes over the "edge" of the tape. To make this formal we define five possible *actions* the Turing machine can perform:

$$\begin{array}{llll} a ::= & W_{Bk} \;\text{(write blank, } Bk\text{)} & \mid L \;\text{(move left)} & \mid Nop \;\text{(do-nothing operation)} \\ \mid & W_{Oc} \;\text{(write occupied, } Oc\text{)} & \mid R \;\text{(move right)} & \end{array}$$

We slightly deviate from the presentation in [2] by using the *Nop* operation; however its use will become important when we formalise halting computations and also universal Turing machines. Given a tape and an action, we can define the following tape updating function:

$$\begin{array}{lcl} update\ (l, r)\ W_{Bk} & \stackrel{def}{=} & (l, Bk::tl\ r) \\ update\ (l, r)\ W_{Oc} & \stackrel{def}{=} & (l, Oc::tl\ r) \\ update\ (l, r)\ L & \stackrel{def}{=} & if\ l = []\ then\ ([], Bk::r)\ else\ (tl\ l, hd\ l::r) \\ update\ (l, r)\ R & \stackrel{def}{=} & if\ r = []\ then\ (Bk::l, [])\ else\ (hd\ r::l, tl\ r) \\ update\ (l, r)\ Nop & \stackrel{def}{=} & (l, r) \end{array}$$

The first two clauses replace the head of the right-list with a new *Bk* or *Oc*, respectively. To see that these two clauses make sense in case where r is the empty list, one has to know that the tail function, *tl*, is defined such that $tl\ [] \stackrel{def}{=} []$ holds. The third clause implements the move of the head one step to the left: we need to test if the left-list l is empty; if yes, then we just prepend a blank cell to the right-list; otherwise we have to remove the head from the left-list and prepend it to the right-list. Similarly in the fourth clause for a right move action. The *Nop* operation leaves the the tape unchanged.

Next we need to define the *states* of a Turing machine. We follow the choice made in [1] representing a state by a natural number and the states in a Turing machine program by the initial segment of natural numbers starting from *0*. In doing so we can compose two Turing machine programs by shifting the states of one by an appropriate amount to a higher segment and adjusting some "next states" in the other.

An *instruction* of a Turing machine is a pair consisting of an action and a natural number (the next state). A *program p* of a Turing machine is then a list of such pairs. Using as an example the following Turing machine program, which consists of four instructions

$$dither \stackrel{def}{=} [\overbrace{\underbrace{(W_{Bk}, 1)}_{Bk\text{-case}}, \overbrace{(R, 2)}^{Oc\text{-case}}}^{\text{1st state = starting state}}, \underbrace{(L, 1), (L, 0)}_{\text{2nd state}}] \tag{1}$$

the reader can see we have organised our Turing machine programs so that segments of two belong to a state. The first component of such a segment determines what action should be taken and which next state should be transitioned to in case the head reads a *Bk*; similarly the second component determines what should be done in case of reading *Oc*. We have the convention that the first state is always the *starting state* of the Turing machine. The *0*-state is special in that it will be used as the "halting state". There are no instructions for the *0*-state, but it will always perform a *Nop*-operation and remain in the *0*-state. Unlike Asperti and Riccioti [1], we have chosen a very concrete representation for programs, because when constructing a universal Turing machine, we need to define a coding function for programs. This can be easily done for our programs-as-lists, but is more difficult for the functions used by Asperti and Ricciotti.

Given a program *p*, a state and the cell being read by the head, we need to fetch the corresponding instruction from the program. For this we define the function *fetch*

$$\begin{array}{l} fetch\ p\ 0\ \_ = (Nop, 0) \\ fetch\ p\ (Suc\ s)\ Bk \stackrel{def}{=} case\ nth\_of\ p\ (2 * s)\ of \\ \qquad\qquad\qquad\qquad None \Rightarrow (Nop, 0) \mid Some\ i \Rightarrow i \\ fetch\ p\ (Suc\ s)\ Oc \stackrel{def}{=} case\ nth\_of\ p\ (2 * s + 1)\ of \\ \qquad\qquad\qquad\qquad None \Rightarrow (Nop, 0) \mid Some\ i \Rightarrow i \end{array} \tag{2}$$

In this definition the function *nth_of* returns the *n*th element from a list, provided it exists (*Some*-case), or if it does not, it returns the default action *Nop* and the default state *0* (*None*-case). We often need to restrict Turing machine programs to be well-formed: a program *p* is *well-formed* if it satisfies the following three properties:

$$wf\ p \stackrel{def}{=} 2 \leq length\ p \wedge iseven\ (length\ p) \wedge (\forall\ (a, s) \in p.\ s \leq length\ p\ div\ 2)$$

The first states that *p* must have at least an instruction for the starting state; the second that *p* has a *Bk* and *Oc* instruction for every state, and the third that every next-state is one of the states mentioned in the program or being the *0*-state.

We need to be able to sequentially compose Turing machine programs. Given our concrete representation, this is relatively straightforward, if slightly fiddly. We use the following two auxiliary functions:

$$\begin{array}{l} \mathit{shift}\ p\ n \stackrel{\mathit{def}}{=} \mathit{map}\ (\lambda(a, s).\ (a, \mathit{if}\ s = 0\ \mathit{then}\ 0\ \mathit{else}\ s + n))\ p \\ \mathit{adjust}\ p \stackrel{\mathit{def}}{=} \mathit{map}\ (\lambda(a, s).\ (a, \mathit{if}\ s = 0\ \mathit{then}\ \mathit{Suc}\ (\mathit{length}\ p\ \mathit{div}\ 2)\ \mathit{else}\ s))\ p \end{array}$$

The first adds n to all states, exept the *0*-state, thus moving all "regular" states to the segment starting at n; the second adds $\mathit{Suc}\ (\mathit{length}\ p\ \mathit{div}\ 2)$ to the *0*-state, thus redirecting all references to the "halting state" to the first state after the program p. With these two functions in place, we can define the *sequential composition* of two Turing machine programs p_1 and p_2 as

$$p_1 \oplus p_2 \stackrel{\mathit{def}}{=} \mathit{adjust}\ p_1\ @\ \mathit{shift}\ p_2\ (\mathit{length}\ p_1\ \mathit{div}\ 2)$$

A *configuration* c of a Turing machine is a state together with a tape. This is written as $(s, (l, r))$. We say a configuration *is final* if $s = 0$ and we say a predicate P *holds for* a configuration if P holds for the tape (l, r). If we have a configuration and a program, we can calculate what the next configuration is by fetching the appropriate action and next state from the program, and by updating the state and tape accordingly. This single step of execution is defined as the function *step*

$$\begin{array}{l} \mathit{step}\ (s, (l, r))\ p \stackrel{\mathit{def}}{=} \mathit{let}\ (a, s') = \mathit{fetch}\ p\ s\ (\mathit{read}\ r) \\ \qquad\qquad\qquad\qquad\quad \mathit{in}\ (s', \mathit{update}\ (l, r)\ a) \end{array}$$

where *read r* returns the head of the list r, or if r is empty it returns *Bk*. It is impossible in Isabelle/HOL to lift the *step*-function to realise a general evaluation function for Turing machines. The reason is that functions in HOL-based provers need to be terminating, and clearly there are Turing machine programs that are not. We can however define an evaluation function that performs exactly n steps:

$$\begin{array}{l} \mathit{steps}\ c\ p\ 0 \stackrel{\mathit{def}}{=} c \\ \mathit{steps}\ c\ p\ (\mathit{Suc}\ n) \stackrel{\mathit{def}}{=} \mathit{steps}\ (\mathit{step}\ c\ p)\ p\ n \end{array}$$

Recall our definition of *fetch* (shown in (2)) with the default value for the *0*-state. In case a Turing program takes according to the usual textbook definition [2] less than n steps before it halts, then in our setting the *steps*-evaluation does not actually halt, but rather transitions to the *0*-state (the final state) and remains there performing *Nop*-actions until n is reached.

tapes in standard form

Before we can prove the undecidability of the halting problem for our Turing machines, we need to analyse two concrete Turing machine programs and establish that they are correct—that means they are "doing what they are supposed to be doing". Such correctness proofs are usually left out in the informal literature, for example [2]. One program we need to prove correct is the *dither* program shown in (1) and the other program is *copy* is defined as

$copy_{begin} \stackrel{def}{=}$
$[(W_{Bk}, 0), (R, 2), (R, 3), (R, 2), (W_{Oc}, 3), (L, 4), (L, 4), (L, 0)]$

$copy_{loop} \stackrel{def}{=}$
$[(R, 0), (R, 2), (R, 3), (W_{Bk}, 2), (R, 3), (R, 4), (W_{Oc}, 5), (R, 4), (L, 6), (L, 5), (L, 6), (L, 1)]$

$copy_{end} \stackrel{def}{=}$
$[(L, 0), (R, 2), (W_{Oc}, 3), (L, 4), (R, 2), (R, 2), (L, 5), (W_{Bk}, 4), (R, 0), (L, 5)]$

Fig. 1. Copy machine

$$copy \stackrel{def}{=} copy_{begin} \oplus copy_{loop} \oplus copy_{end}$$

whose three components are given in Figure 1. To the prove correctness of these Turing machine programs, we introduce the notion of total correctness defined in terms of *Hoare-triples*, written $\{P\}\ p\ \{Q\}$. They realise the idea that a program p started in state 1 with a tape satisfying P will after n steps halt (have transitioned into the halting state) with a tape satisfying Q. We also have *Hoare-pairs* of the form $\{P\}\ p \uparrow$ realising the case that a program p started with a tape satisfying P will loop (never transition into the halting state). Both notion are formally defined as

$\{P\}\ p\ \{Q\} \stackrel{def}{=}$
$\forall\ (l, r).$
if $P\ (l, r)$ holds then
$\exists\ n.$ such that
$is\_final\ (steps\ (1, (l, r))\ p\ n)\ \wedge$
$Q\ holds\_for\ (steps\ (1, (l, r))\ p\ n)$

$\{P\}\ p \uparrow \stackrel{def}{=}$
$\forall\ (l, r).$
if $P\ (l, r)$ holds then
$\forall\ n. \neg\ is\_final\ (steps\ (1, (l, r))\ p\ n)$

We have set up our Hoare-style reasoning so that we can deal explicitly with looping and total correctness, rather than have notions for partial correctness and termination. Although the latter would allow us to reason more uniformly (only using Hoare-triples), we prefer our definitions because we can derive simple Hoare-rules for sequentially composed Turing programs. In this way we can reason about the correctness of $copy_{begin}$, for example, completely separately from $copy_{loop}$.

It is rather straightforward to prove that the Turing program *dither* satisfies the following correctness properties

$$\{dither\_halt\_inv\}\ dither\ \{dither\_halt\_inv\}$$
$$\{dither\_unhalt\_inv\}\ dither \uparrow$$

unfold The first states that on a tape $(Bk \uparrow n, [Oc, Oc])$ halts in tree steps leaving the tape unchanged. In the other states that *dither* started with tape $(Bk \uparrow n, [Oc])$ loops.

In the following we will consider the following Turing machine program that makes a copies a value on the tape.

assertion holds for all tapes

Hoare rule for composition

For showing the undecidability of the halting problem, we need to consider two specific Turing machines. copying TM and dithering TM

correctness of the copying TM
measure for the copying TM, which we however omit.
halting problem

$$\frac{\{P_1\}\, p_1\, \{Q_1\} \quad \{P_2\}\, p_2\, \{Q_2\} \quad Q_1 \mapsto P_2 \quad wf\, p_1}{\{P_1\}\, p_1 \oplus p_2\, \{Q_2\}} \qquad \frac{\{P_1\}\, p_1\, \{Q_1\} \quad \{P_2\}\, p_2 \uparrow \quad Q_1 \mapsto P_2 \quad wf\, p_1}{\{P_1\}\, p_1 \oplus p_2 \uparrow}$$

3 Abacus Machines

Boolos et al [2] use abacus machines as a stepping stone for making it less laborious to write programs for Turing machines. Abacus machines operate over an unlimited number of registers $R_0, R_1, \ldots$ each being able to hold an arbitrary large natural number. We use natural numbers to refer to registers, but also to refer to *opcodes* of abacus machines. Obcodes are given by the datatype

$$\begin{array}{rll} o ::= & \mathit{Inc}\ R & \text{increment register } R \text{ by one} \\ | & \mathit{Dec}\ R\ o & \text{if content of } R \text{ is non-zero,} \\ & & \text{then decrement it by one} \\ & & \text{otherwise jump to opcode } o \\ | & \mathit{Goto}\ o & \text{jump to opcode } o \end{array}$$

A *program* of an abacus machine is a list of such obcodes. For example the program clearing the register R (setting it to 0) can be defined as follows:

The second opcode *Goto* $(0::'a)$ in this program means we jump back to the first opcode, namely *Dec R o*. The *memory* m of an abacus machine holding the values of the registers is represented as a list of natural numbers. We have a lookup function for this memory, written *abc_lm_v m R*, which looks up the content of register R; if R is not in this list, then we return 0. Similarly we have a setting function, written *abc_lm_s m R n*, which sets the value of R to n, and if R was not yet in m it pads it approriately with 0s.

Abacus machine halts when it jumps out of range.

4 Recursive Functions

5 Wang Tiles

Used in texture mapings - graphics

6 Related Work

The most closely related work is by Norrish [4], and Asperti and Ricciotti [1]. Norrish bases his approach on lambda-terms. For this he introduced a clever rewriting technology based on combinators and de-Bruijn indices for rewriting modulo β-equivalence (to keep it manageable)

Given some input tape (l_i,r_i), we can define when a program p generates a specific output tape (l_o,r_o)

$$\begin{array}{l} \mathit{runs}\ p\ (l_i, r_i)\ (l_o, r_o) \stackrel{\mathit{def}}{=} \\ \quad \exists n.\ \mathit{nsteps}\ (1, (l_i,r_i))\ p\ n = (0, (l_o,r_o)) \end{array}$$

where *1* stands for the starting state and *0* for our final state. A program p with input tape (l_i, r_i) *halts* iff

$$\mathit{halts}\ p\ (l_i, r_i) \stackrel{\mathit{def}}{=} \exists l_o\ r_o.\ \mathit{runs}\ p\ (l_i, r_i)\ (l_o, r_o)$$

Later on we need to consider specific Turing machines that start with a tape in standard form and halt the computation in standard form. To define a tape in standard form, it is useful to have an operation that translates lists of natural numbers into tapes.

By this we mean

This means the Turing machine starts with a tape containg *n Oc*s and the head pointing to the first one; the Turing machine halts with a tape consisting of some *Bk*s, followed by a "cluster" of *Oc*s and after that by some *Bk*s. The head in the output is pointing again at the first *Oc*. The intuitive meaning of this definition is to start the Turing machine with a tape corresponding to a value *n* and producing a new tape corresponding to the value *l* (the number of *Oc*s clustered on the output tape).

References

1. A. Asperti and W. Ricciotti. Formalizing Turing Machines. In *Proc. of the 19th International Workshop on Logic, Language, Information and Computation (WoLLIC)*, volume 7456 of *LNCS*, pages 1–25, 2012.
2. G. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and Logic (5th ed.)*. Cambridge University Press, 2007.
3. E. W. Dijkstra. Go to Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968.
4. M. Norrish. Mechanised Computability Theory. In *Proc. of the 2nd Conference on Interactive Theorem Proving (ITP)*, volume 6898 of *LNCS*, pages 297–311, 2011.
5. E. Post. Finite Combinatory Processes-Formulation 1. *Journal of Symbolic Logic*, 1(3):103–105, 1936.
6. C. Urban, J. Cheney, and S. Berghofer. Mechanizing the Metatheory of LF. *ACM Transactions on Computational Logic*, 12:15:1–15:42, 2011.
7. C. Wu, X. Zhang, and C. Urban. ??? Submitted, 2012.