Formalising 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. Our theory can be used to formalise other computability results: we give one example about the undecidability of Wang's tiling problem, whose proof uses the notion of a universal Turing machine.

1 Introduction

We formalised in earlier work the correctness proofs for two algorithms in Isabelle/HOL—one about type-checking in LF [**?**] and another about deciding requests in access control [**?**]. The formalisations uncovered a gap in the informal correctness proof of the former and made us realise that important details were left out in the informal model for the latter. However, in both cases we were unable to formalise in Isabelle/HOL computability arguments about the algorithms. The reason is that both algorithms are formulated in terms of inductive predicates. Suppose P stands for one such predicate. Decidability of P usually amounts to showing whether $P \vee \neg P$ holds. But this does *not* work in Isabelle/HOL, since it is a theorem prover 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. The same problem would arise if we had formulated the algorithms as recursive functions, because internally in Isabelle/HOL, like in all HOL-based theorem provers, functions are represented as inductively defined predicates too.

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 HOL4 theorem prover. He choose the λ-calculus as the starting point for his formalisation of computability theory, because of its "simplicity" [**?**, 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 in the context of theorem provers [**?**, 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 interactive theorem provers, like Isabelle/HOL, are at their best when the data-structures at hand are "structurally" defined, like lists, natural numbers, regular expressions, etc. Such data-structures come with convenient reasoning infrastructures (for example induction principles, recursion combinators and so on). But this is *not* the case with Turing machines (and also not with register machines): underlying their definitions are sets of states together with transition functions, all of which are not structurally defined. This means we have to implement our own reasoning infrastructure in order to prove properties about them. This leads to annoyingly fiddly formalisations. We noticed first the difference between both, structural and non-structural, "worlds" when formalising the Myhill-Nerode theorem, where regular expressions fared much better than automata [**?**]. However, with Turing machines there seems to be no alternative if one wants to formalise the great many proofs from the literature that use them. We will analyse one example—undecidability of Wang's tiling problem—in Section 5. The standard proof of this property uses the notion of universal Turing machines.

We are not the first who formalised Turing machines in a theorem prover: we are aware of the preliminary work by Asperti and Ricciotti [**?**]. 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" [**?**, Page 2]. For our formalisation we followed mainly the proofs from the textbook [**?**] 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 [**?**]. However, one aspect that is completely left out from the informal description in [**?**], and similar ones we are aware of, is arguments why certain Turing machines are correct. We will introduce Hoare-style proof rules which help us with such correctness arguments of Turing machines.

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 [**?**, Page 23]:

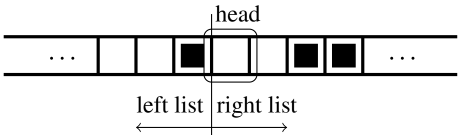
> *"In particular, the fact that the universal machine operates with a different alphabet with respect to the machines it simulates is annoying."*

In this paper we follow the approach by Boolos et al [**?**], which goes back to Post [**?**], where all Turing machines operate on tapes that contain only *blank* or *occupied* cells (represented by *Bk* and *Oc*, respectively, in our formalisation). 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 [**?**] 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 [**?**], instead follow the proof in [**?**] by relating 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 [**?**] 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. We construct the universal Turing machine from [**?**] by relating 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 the undecidability of Wang's tiling problem. We are not aware of any other formalisation of a substantial undecidability problem.

2 Turing Machines

Turing machines can be thought of as having a read-write-unit, also referred to as *head*, "gliding" over a potentially infinite tape. Boolos et al [**?**] 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 operates. 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:

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

We slightly deviate from the presentation in [**?**] 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:

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

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 in Isabelle/HOL such that $\mathit{tl}\ [] \overset{def}{=} []$ 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 (last clause).

Note that our treatment of the tape is rather "unsymmetric"—we have the convention that the head of the right-list is where the head is currently positioned. Asperti and Ricciotti [**?**] also considered such a representation, but dismiss it as it complicates their definition for *tape equality*. The reason is that moving the head one step to the left and then back to the right might change the tape (in case of going over the "edge"). Therefore they distinguish four types of tapes: one where the tape is empty; another where the head is on the left edge, respectively right edge, and in the middle of the tape. The reading, writing and moving of the tape is then defined in terms of these four cases. In this way they can keep the tape in a "normalised" form, and thus making a left-move followed by a right-move being the identity on tapes. Since we are not using the notion of tape equality, we can get away with the unsymmetric definition above, and by using the *update* function cover uniformly all cases including corner cases.

Next we need to define the *states* of a Turing machine. Given how little is usually said about how to represent them in informal presentations, it might be surprising that in a theorem prover we have to select carefully a representation. If we use the naive representation where a Turing machine consists of a finite set of states, then we will have difficulties composing two Turing machines: we would need to combine two finite sets of states, possibly renaming states apart whenever both machines share states.[3] This renaming can be quite cumbersome to reason about. Therefore we made the choice of representing a state by a natural number and the states of a Turing machine will always consist of the initial segment of natural numbers starting from *0* up to the number of

[3] The usual disjoint union operation in Isabelle/HOL cannot be used as it does not preserve types.

states of the machine. In doing so we can compose two Turing machine by shifting the states of one by an appropriate amount to a higher segment and adjusting some "next states" in the other.

An *instruction i* 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

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

the reader can see we have organised our Turing machine programs so that segments of two belong to a state. The first component of the 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 zeroth 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 [**?**], 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*

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

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). In doing so we slightly deviate from the description in [**?**]: if their Turing machines transition to a non-existing state, then the computation is halted. We will transition in such cases to the *0*-state. However, with introducing the notion of *well-formed* Turing machine programs we will later exclude such cases and make the *0*-state the only "halting state". A program *p* is said to be well-formed if it satisfies the following three properties:

$$\begin{aligned} \textit{t\_correct}\ p &\overset{def}{=} 2 \leq \textit{length}\ p\\ &\wedge\ \textit{iseven}\ (\textit{length}\ p)\\ &\wedge\ \forall\,(a,\,s)\in p.\ s \leq \textit{length}\ p\ \textit{div}\ 2 \end{aligned}$$

The first says 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.

A *configuration c* of a Turing machine is a state together with a tape. This is written as $(s,\,(l,\,r))$. 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 *tstep*

$$\begin{aligned} &\textit{step}\ (s,\,(l,\,r))\ p \overset{def}{=}\\ &\qquad \textit{let}\ (a,\,s) = \textit{fetch}\ p\ s\ (\textit{read}\ r)\\ &\qquad \textit{in}\ (s',\,\textit{update}\ (l,\,r)\ a) \end{aligned}$$

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 realising 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 so that it performs exactly *n* steps:

$$\begin{aligned} &\textit{steps}\ c\ p\ 0 &&\overset{def}{=} c\\ &\textit{steps}\ c\ p\ (\textit{Suc}\ n) &&\overset{def}{=} \textit{steps}\ (\textit{step}\ c\ p)\ p\ n \end{aligned}$$

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

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

$$\begin{aligned} &\textit{runs}\ p\ (l_i,\,r_i)\ (l_o,\,r_o) \overset{def}{=}\\ &\qquad \exists\,n.\ \textit{nsteps}\ (1,\,(l_i,r_i))\ p\ n = (0,\,(l_o,r_o)) \end{aligned}$$

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

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

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).

Before we can prove the undecidability of the halting problem for Turing machines, we have to define how to compose two Turing machines. Given our setup, this is relatively straightforward, if slightly fiddly. We use the following two auxiliary functions:

$$\begin{array}{l} \mathit{shift}\ p\ n \overset{\mathit{def}}{=} \\ \quad \mathit{map}\ (\lambda(a,\, s).\ (a,\, \mathit{if}\ s = \mathit{0}\ \mathit{then}\ \mathit{0}\ \mathit{else}\ s + n))\ p \\ \mathit{turing\_basic.adjust}\ p \overset{\mathit{def}}{=} \\ \quad \mathit{map}\ (\lambda\ (a,\, s). \\ \qquad\qquad (a,\, \mathit{if}\ s = \mathit{0}\ \mathit{then}\ \mathit{length}\ p\ \mathit{div}\ \mathit{2} + \mathit{1}\ \mathit{else}\ s))\ p \end{array}$$

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 *length p div 2* + *1* to the *0*-state, thus ridirecting 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

$$\mathit{p1.1} \oplus \mathit{p2.1} \overset{\mathit{def}}{=} \mathit{turing\_basic.adjust}\ \mathit{p1.1}\ @\ \mathit{shift}\ \mathit{p2.1}\ (\mathit{length}\ \mathit{p1.1}\ \mathit{div}\ \mathit{2})$$

This means p_1 is executed first. Whenever it originally transitioned to the *0*-state, it will in the composed program transition to the starting state of p_2 instead. All the states of p_2 have been shifted in order to make sure that the states of the composed program p_1 $\oplus$ p_2 still only "occupy" an initial segment of the natural numbers.

$$\mathit{copy} \overset{\mathit{def}}{=} \mathit{tcopy\_init} \oplus \mathit{tcopy\_loop} \oplus \mathit{tcopy\_end}$$

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

3 Abacus Machines

Boolos et al [**?**] 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

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

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* $(\mathit{0}::'a)$ in this programm 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 [**?**], and Asperti and Ricciotti [**?**]. 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)

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. M. Norrish. Mechanised Computability Theory. In *Proc. of the 2nd Conference on Interactive Theorem Proving (ITP)*, volume 6898 of *LNCS*, pages 297–311, 2011.
4. E. Post. Finite Combinatory Processes-Formulation 1. *Journal of Symbolic Logic*, 1(3):103–105, 1936.
5. C. Urban, J. Cheney, and S. Berghofer. Mechanizing the Metatheory of LF. *ACM Transactions on Computational Logic*, 12:15:1–15:42, 2011.
6. C. Wu, X. Zhang, and C. Urban. A Formalisation of the Myhill-Nerode Theorem based on Regular Expressions (Proof Pearl). In *Proc. of the 2nd Conference on Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 341–356, 2011.
7. C. Wu, X. Zhang, and C. Urban. ??? Submitted, 2012.