Formalising Computability Theory in Isabelle/HOL

Jian Xu, Xingyuan Zhang
PLA University of Science and Technology Nanjing, China

Christian Urban
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.**

***Keywords*-Turing Machines, Computability, Isabelle/HOL, Wang tilings**

I. Introduction

We formalised in earlier work the correctness proofs for two algorithms in Isabelle/HOL—one about type-checking in LF [5] and another about deciding requests in access control [7]. 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 $P \lor \neg P$ 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 $P \lor \neg P$ 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" [3, 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 [3, Page 310]:

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

In this paper we took 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 definition is a set of states together with a transition function, both 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 [6]. 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 V. 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 [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 useable as a guideline for formalization" [1, Page 2]. For our formalisation we followed 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 generalize 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]. However, one aspect that is completely left out from the informal description in [2], and similar ones we are aware of, are 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 [1, Page 23]:

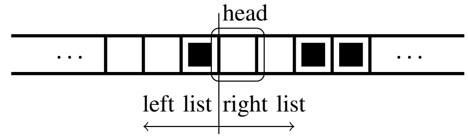
> *"In particular, the fact that the universal machine operates with a different alphabet with respect to the machines it simulates is annoying."*

In this paper we follow the approach by Boolos et al [2], which goes back to Post [4], 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 [2] makes the reasoning more uniform. In addition some proofs *about* Turing machines will be 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. Therefore in order to construct a *universal Turing machine* we follow the proof in [2] 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:

II. Turing Machines

Turing machines can be thought of as having a read-write-unit "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 read-write-unit and r for the tape on the right-hand side. We have the convention that the head, written *hd*, of the right-list is the cell on which the read-write-unit 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 cell whenever the read-write unit goes over the "edge" of the tape. To make this formal we define five possible *actions* the Turing machine can perform:

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

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

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

The first two clauses replace the head of the right-list with new *Bk* or *Oc*, repsectively. To see that these 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 read-write unit to the left: we need to test if the left-list is empty; if yes, then we just add a blank cell to the right-list; otherwise we have to remove the head from the left-list and add it to the right-list. Similarly in the clause for the right move. The *Nop* operation leaves the the tape unchanged.

Note that our treatment of the tape is rather "unsymmetric"—we have the convention that the head of the right-list is where the read-write unit is currently possitioned. Asperti and Ricciotti [1] also consider such a representation, but dismiss it as it complicates their definition for *tape equality*. The reason is that moving the read-write unit to the left and then back to the right can change the tape (in case of going over the "edge"). Therefore they distinguish four cases where the tape is empty as well as the read-write unit on the left edge, respectively right edge, or in the middle. The reading and moving of the tape is then defined in terms of these four cases. Since we are not going to use the notion of tape equality, we can get away with the definition above and be done with all corner cases.

Next we need to define the *states* of a Turing machine. Given how little is usually said about how to represent states in informal presentaions, it might be surprising that in a theorem prover we have to select carfully a representation. If we use the naive representation as a Turing machine consiting of a finite set of states, then we will have difficulties composing two Turing machines. We would need to somehow combine the two finite sets of states, possibly renaming states apart if both machines share states. 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 number of states of the machine minus *1*. In doing so we can compose two Turing machine by "shifting" the states of one by an appropriate amount to a higher segment.

An *instruction* of a Turing machine is a pair consisting of a natural number (the next state) and an action. A *program* of a Turing machine is then a list of such pairs. Given a program *p*, a state and a cell being read by the read-write unnit, we need to fetch the corresponding instruction in the program. For this we define the function *fetch*

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

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

No evaluator in HOL, because of totality.

III. Abacus Machines

IV. Recursive Functions

V. Wang Tiles

Used in texture mapings - graphics

VI. Related Work

The most closely related work is by Norrish [3], 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)

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.