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. Following the textbook by Boolos et al, we formalise Turing machines and relate them to abacus machines and recursive functions. We "tie the know" between these three computational models by formalising a universal function and obtaining from it a universal Turing machine by our verified translation from recursive functions to abacus programs and from abacus programs to Turing machine programs. Hoare-style reasoning techniques allow us to reason about Turing machine and abacus programs. Our theory can be used to formalise other computability results.

1 Introduction

The motivation of this paper is to bring the ability to reason about results from computability theory, for example decidability of the halting problem, to the theorem prover Isabelle/HOL. Norrish formalised computability theory in HOL4. He choose the λ-calculus as the starting point for his formalisation because of its "simplicity" [8, 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 [8, 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 (similarly abacus programs) can be completely *unstructured*, behaving similar to Basic programs containing 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 concrete Turing machine programs is usually left out in the informal literature, e.g. [2].

We are not the first who formalised Turing machines: we are aware of the work by Asperti and Ricciotti [1]. They describe a complete formalisation of Turing machines in the Matita theorem prover, including a universal Turing machine. However, they do *not* formalise the undecidability of the halting problem since their main focus is complexity, rather than computability theory. They also 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 by Boolos et al [2] and found that the description there is quite detailed. Some details are left out however: for example, constructing the *copy Turing machine* is left as an exercise to the reader—a corresponding correctness proof is not mentioned at all; also [2] only shows 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."*

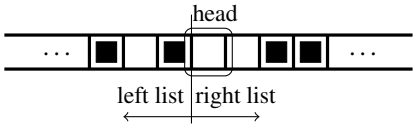
In this paper we follow the approach by Boolos et al [2], which goes back to Post [9], where all Turing machines operate on tapes that contain only *blank* or *occupied* cells. Traditionally the content of a cell can be any character from a finite alphabet. Although computationally equivalent, the more 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 by translating from a recursive function. The part of mechanising the translation of recursive function to register machines has already been done by Zammit in HOL [11], although his register machines use a slightly different instruction set than the one described in [2].

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. This works essentially like a small, verified compiler from recursive functions to Turing machine

programs. When formalising the universal Turing machine, we stumbled in [2] upon an inconsistent use of the definition of what partial function a Turing machine calculates.

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 use the notation Bk^n (similarly Oc^n) for lists composed of n elements of Bks. We also have the convention that the head, abbreviated *hd*, of the right list is the cell on which the head of the Turing machine currently scans. 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:

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

We slightly deviate from the presentation in [2] (and also [1]) 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 tape unchanged.

Next we need to define the *states* of a Turing machine. We follow the choice made in [1] by representing a state with 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

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

the reader can see we have organised our Turing machine programs so that segments of two pairs 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. We have chosen a very concrete representation for Turing machine programs, because when constructing a universal Turing machine, we need to define a coding function for programs.

Given a program *p*, a state and the cell being scanned 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 have to restrict Turing machine programs to be well-formed: a program *p* is *well-formed* if it satisfies the following three properties:

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

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.

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*

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

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 in order to realise a general evaluation function for Turing machines programs. The reason is that functions in HOL-based provers need to be terminating, and clearly there are programs that are not.[3] We can however define a recursive 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, say [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.

We often need to restrict tapes to be in standard form, which means the left list of the tape is either empty or only contains Bks, and the right list contains some "clusters" of Ocs separated by single blanks. To make this formal we define the following overloaded function encoding natural numbers into lists of Ocs and Bks.

$$\begin{array}{lll} \langle n \rangle \stackrel{\mathit{def}}{=} Oc^n + 1 & & \langle [] \rangle \stackrel{\mathit{def}}{=} [] \\ \langle (n, m) \rangle \stackrel{\mathit{def}}{=} \langle n \rangle\ @\ [Bk]\ @\ \langle m \rangle & & \langle [n] \rangle \stackrel{\mathit{def}}{=} \langle n \rangle \\ & & \langle n::ns \rangle \stackrel{\mathit{def}}{=} \langle (n, ns) \rangle \end{array} \tag{3}$$

A *standard tape* is then of the form $(Bk^k, \langle [n_1,\ldots,n_m] \rangle\ @\ Bk^l)$ for some k, l and $n_{1\ldots m}$. Note that the head in a standard tape "points" to the leftmost Oc on the tape. Note also that the natural number *0* is represented by a single filled cell on a standard tape, *1* by two filled cells and so on.

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:

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

[3] There is the alternative to use partial functions, which do not necessarily need to terminate, but this does not provide us with a useful induction principle [5].

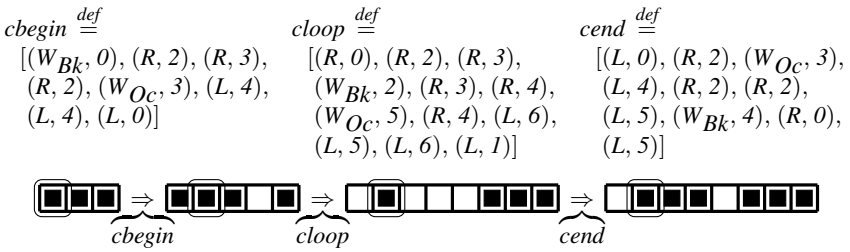


Fig. 1. The three components of the *copy Turing machine* (above). If started (below) with the tape $([], \langle 2 \rangle)$ the first machine appends $[Bk, Oc]$ at the end of the right tape; the second then "moves" all Ocs except the first from the beginning of the tape to the end; the third "refills" the original block of Ocs. The resulting tape is $([Bk], \langle (2, 2) \rangle)$.

The first adds n to all states, except the *0*-state, thus moving all "regular" states to the segment starting at n; the second adds $Suc\ (length\ p\ div\ 2)$ 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

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

Before we can prove the undecidability of the halting problem for our Turing machines working on standard tapes, we have 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]. The first program we need to prove correct is the *dither* program shown in (1) and the second program is *copy* defined as

$$copy \stackrel{def}{=} cbegin\ ;\ cloop\ ;\ cend \tag{4}$$

whose three components are given in Figure 1. For our correctness proofs, we introduce the notion of total correctness defined in terms of *Hoare-triples*, written $\{P\}\ p\ \{Q\}$. They implement the idea that a program p started in state *1* with a tape satisfying P will after some n steps halt (have transitioned into the halting state) with a tape satisfying Q. This idea is very similar to the notion of *realisability* in [1]. We also have *Hoare-pairs* of the form $\{P\}\ p \uparrow$ implementing 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

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

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

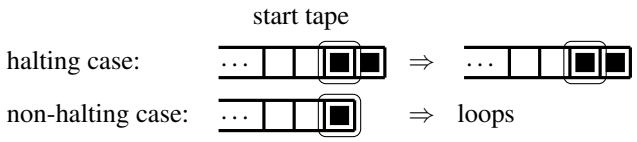
For our Hoare-triples we can easily prove the following Hoare-consequence rule

$$\frac{P' \mapsto P \quad \{P\}\, p\, \{Q\} \quad Q \mapsto Q'}{\{P'\}\, p\, \{Q'\}} \tag{5}$$

where $P' \mapsto P$ stands for the fact that for all tapes tp, $P'\, tp$ implies $P\, tp$ (similarly for Q and Q').

Like Asperti and Ricciotti with their notion of realisability, we have set up our Hoare-rules so that we can deal explicitly with total correctness and non-termination, 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 below some simple Hoare-rules for sequentially composed Turing programs. In this way we can reason about the correctness of *cbegin*, for example, completely separately from *cloop* and *cend*.

It is relatively straightforward to prove that the Turing program *dither* shown in (1) is correct. This program should be the "identity" when started with a standard tape representing *1* but loops when started with the *0*-representation instead, as pictured below.



We can prove the following Hoare-statements:

$$\{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 1\rangle)\}\ dither\ \{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 1\rangle)\}$$
$$\{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 0\rangle)\}\ dither \uparrow$$

The first is by a simple calculation. The second is by an induction on the number of steps we can perform starting from the input tape.

The program *copy* defined in (4) has 15 states; its purpose is to produce the standard tape $(Bks, \langle (n, n)\rangle)$ when started with $(Bks, \langle n\rangle)$, that is making a copy of a value n on the tape. Reasoning about this program is substantially harder than about *dither*. To ease the burden, we derive the following two Hoare-rules for sequentially composed programs.

$$\frac{\{P\}\, p_1\, \{Q\} \quad \{Q\}\, p_2\, \{R\}}{\{P\}\, p_1\, ;\, p_2\, \{R\}}\ wf\, p_1 \qquad \frac{\{P\}\, p_1\, \{Q\} \quad \{Q\}\, p_2 \uparrow}{\{P\}\, p_1\, ;\, p_2 \uparrow}\ wf\, p_1$$

The first corresponds to the usual Hoare-rule for composition of two terminating programs. The second rule gives the conditions for when the first program terminates generating a tape for which the second program loops. The side-conditions about $wf\, p_1$ are needed in order to ensure that the redirection of the halting and initial state in p_1 and p_2, respectively, match up correctly. These Hoare-rules allow us to prove the correctness of *copy* by considering the correctness of the components *cbegin*, *cloop* and *cend* in isolation. This simplifies the reasoning considerably, for example when designing

$$
\begin{array}{lcll}
I_1\ n\ (l, r) & \stackrel{def}{=} & (l, r) = ([], Oc^n) & \text{(starting state)} \\
I_2\ n\ (l, r) & \stackrel{def}{=} & \exists\, i\, j.\ 0 < i \wedge i + j = n \wedge (l, r) = (Oc^i, Oc^j) & \\
I_3\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, tl\ r) = (Bk::Oc^n, []) & \\
I_4\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, r) = (Oc^n, [Bk, Oc]) \vee (l, r) = (Oc^{n-1}, [Oc, Bk, Oc]) & \\
I_0\ n\ (l, r) & \stackrel{def}{=} & 1 < n \wedge (l, r) = (Oc^{n-2}, [Oc, Oc, Bk, Oc]) \vee & \text{(halting state)} \\
 & & n = 1 \wedge (l, r) = ([], [Bk, Oc, Bk, Oc]) & \\
\hline
J_1\ n\ (l, r) & \stackrel{def}{=} & \exists\, i\, j.\ i + j + 1 = n \wedge (l, r) = (Oc^i, Oc::Oc::Bk^j\ @\ Oc^j) \wedge 0 < j \vee & \\
 & & 0 < n \wedge (l, r) = ([], Bk::Oc::Bk^n\ @\ Oc^n) & \text{(starting state)} \\
J_0\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, r) = ([Bk], Oc::Bk^n\ @\ Oc^n) & \text{(halting state)} \\
\hline
K_1\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, r) = ([Bk], Oc::Bk^n\ @\ Oc^n) & \text{(starting state)} \\
K_0\ n\ (l, r) & \stackrel{def}{=} & 0 < n \wedge (l, r) = ([Bk], Oc^n\ @\ (Bk::Oc^n)) & \text{(halting state)}
\end{array}
$$

Fig. 2. The invariants $I_0,\ldots,I_4$ are for the states of *cbegin*. Below, the invariants only for the starting and halting states of *cloop* and *cend* are shown. In each invariant, the parameter n stands for the number of Ocs with which the Turing machine is started.

decreasing measures for proving the termination of the programs. We will show the details for the program *cbegin*. For the two other programs we refer the reader to our formalisation.

Given the invariants $I_0,\ldots,I_4$ shown in Figure 2, which correspond to each state of *cbegin*, we define the following invariant for the whole *cbegin* program:

$$
\begin{array}{lcl}
I_{cbegin}\ n\ (s, tp) & \stackrel{def}{=} & \textit{if } s = 0 \textit{ then } I_0\ n\ tp \\
 & & \textit{else if } s = 1 \textit{ then } I_1\ n\ tp \\
 & & \textit{else if } s = 2 \textit{ then } I_2\ n\ tp \\
 & & \textit{else if } s = 3 \textit{ then } I_3\ n\ tp \\
 & & \textit{else if } s = 4 \textit{ then } I_4\ n\ tp \\
 & & \textit{else False}
\end{array}
$$

This invariant depends on n representing the number of Ocs on the tape. It is not hard (26 lines of automated proof script) to show that for $0 < n$ this invariant is preserved under the computation rules *step* and *steps*. This gives us partial correctness for *cbegin*.

We next need to show that *cbegin* terminates. For this we introduce lexicographically ordered pairs (n, m) derived from configurations $(s, (l, r))$ whereby n is the state s, but ordered according to how *cbegin* executes them, that is $1 > 2 > 3 > 4 > 0$; in order to have a strictly decreasing measure, m takes the data on the tape into account and is calculated according to the following measure function:

$$
\begin{array}{lcl}
M_{cbegin}(s, (l, r)) & \stackrel{def}{=} & \textit{if } s = 2 \textit{ then length } r \\
 & & \textit{else if } s = 3 \textit{ then } (\textit{if } r = [] \vee r = [Bk] \textit{ then 1 else 0}) \\
 & & \textit{else if } s = 4 \textit{ then length } l \\
 & & \textit{else 0}
\end{array}
$$

With this in place, we can show that for every starting tape of the form $([], Oc^n)$ with $0 < n$, the Turing machine *cbegin* will eventually halt (the measure decreases in each step). Taking this and the partial correctness proof together, we obtain the Hoare-triple shown on the left for *cbegin*:

$$\{I_1\ n\}\ cbegin\ \{I_0\ n\} \qquad \{J_1\ n\}\ cloop\ \{J_0\ n\} \qquad \{K_1\ n\}\ cend\ \{K_0\ n\}$$

where we assume $0 < n$ (similar reasoning is needed for the Hoare-triples for *cloop* and *cend*). Since the invariant of the halting state of *cbegin* implies the invariant of the starting state of *cloop*, that is $I_0\ n \mapsto J_1\ n$ holds, and also $J_0\ n = K_1\ n$, we can derive the following Hoare-triple for the correctness of *copy*:

$$\{\lambda tp.\ tp = ([], \langle n\rangle)\}\ copy\ \{\lambda tp.\ tp = ([Bk], \langle(n, n)\rangle)\}$$

That means if we start with a tape of the form $([], \langle n\rangle)$ then *copy* will halt with the tape $([Bk], \langle(n, n)\rangle)$, as desired.

Finally, we are in the position to prove the undecidability of the halting problem. A program p started with a standard tape containing the (encoded) numbers *ns* will *halt* with a standard tape containing a single (encoded) number is defined as

$$halts\ p\ ns \stackrel{def}{=} \{\lambda tp.\ tp = ([], \langle ns\rangle)\}\ p\ \{\lambda tp.\ \exists k\ n\ l.\ tp = (Bk^k, \langle n\rangle\ @\ Bk^l)\}$$

This roughly means we considering only Turing machine programs representing functions that take some numbers as input and produce a single number as output. For undecidability, the property we are proving is that there is no Turing machine that can decide in general whether a Turing machine program halts (answer either *0* for halting or *1* for looping). Given our correctness proofs for *dither* and *copy* shown above, this non-existence is now relatively straightforward to establish. We first assume there is a coding function, written *code M*, which represents a Turing machine M as a natural number. No further assumptions are made about this coding function. Suppose a Turing machine H exists such that if started with the standard tape $([Bk], \langle(code\ M, ns)\rangle)$ returns *0*, respectively *1*, depending on whether M halts or not when started with the input tape containing $\langle ns\rangle$. This assumption is formalised as follows—for all M and all lists of natural numbers *ns*:

$$halts\ M\ ns \text{ implies } \{\lambda tp.\ tp = ([Bk], \langle(code\ M, ns)\rangle)\}\ H\ \{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 0\rangle)\}$$
$$\neg\ halts\ M\ ns \text{ implies } \{\lambda tp.\ tp = ([Bk], \langle(code\ M, ns)\rangle)\}\ H\ \{\lambda tp.\ \exists k.\ tp = (Bk^k, \langle 1\rangle)\}$$

The contradiction can be derived using the following Turing machine

$$contra \stackrel{def}{=} copy\ ;\ H\ ;\ dither$$

Suppose *halts contra* [*code contra*] holds. Given the invariants $P_1 \ldots P_3$ shown on the left, we can derive the following Hoare-pair for *contra* on the right.

$$P_1 \stackrel{def}{=} \lambda tp.\ tp = ([], \langle code\ contra\rangle)$$
$$P_2 \stackrel{def}{=} \lambda tp.\ tp = ([Bk], \langle(code\ contra, code\ contra)\rangle)$$
$$P_3 \stackrel{def}{=} \lambda tp.\ \exists k.\ tp = (Bk^k, \langle 0\rangle)$$

$$\dfrac{\dfrac{\{P_1\}\ copy\ \{P_2\} \quad \{P_2\}\ H\ \{P_3\}}{\{P_1\}\ copy\ ;\ H\ \{P_3\}} \qquad \{P_3\}\ dither\uparrow}{\{P_1\}\ contra\uparrow}$$

This Hoare-pair contradicts our assumption that *contra* started with $\langle code\ contra\rangle$ halts.

Suppose $\neg$ *halts contra* [*code contra*] holds. Again, given the invariants $Q_1 \ldots Q_3$ shown on the left, we can derive the Hoare-triple for *contra* on the right.

$$Q_1 \stackrel{def}{=} \lambda tp.\ tp = ([], \langle code\ contra\rangle)$$
$$Q_2 \stackrel{def}{=} \lambda tp.\ tp = ([Bk], \langle (code\ contra, code\ contra)\rangle)$$
$$Q_3 \stackrel{def}{=} \lambda tp.\ \exists k.\ tp = (Bk^k, \langle 1\rangle)$$

$$\dfrac{\dfrac{\{Q_1\}\ copy\ \{Q_2\} \quad \{Q_2\}\ H\ \{Q_3\}}{\{Q_1\}\ copy\ ;\ H\ \{Q_3\}} \qquad \{Q_3\}\ dither\ \{Q_3\}}{\{Q_1\}\ contra\ \{Q_3\}}$$

This time the Hoare-triple states that *contra* terminates with the "output" $\langle 1\rangle$. In both cases we come to a contradiction, which means we have to abandon our assumption that there exists a Turing machine H which can in general decide whether Turing machines terminate.

3 Abacus Machines

Boolos et al [2] use abacus machines as a stepping stone for making it less laborious to write Turing machine programs. Abacus machines operate over a set of registers R_0, R_1, ..., R_n each being able to hold an arbitrary large natural number. We use natural numbers to refer to registers; we also use a natural number to represent a program counter and to represent jumping "addresses", for which we use the letter l. An abacus program is a list of *instructions* defined by the datatype:

$i ::=$	*Inc R*	increment register R by one
$\mid$	*Dec R l*	if content of R is non-zero, then decrement it by one otherwise jump to instruction l
$\mid$	*Goto l*	jump to instruction l

For example the program clearing the register R (that is setting it to *0*) can be defined as follows:

$$clear\ R\ l \stackrel{def}{=} [Dec\ R\ l, Goto\ 0]$$

Running such a program means we start with the first instruction then execute one instructions after the other, unless there is a jump. For example the second instruction *Goto 0* above means we jump back to the first instruction thereby closing the loop. Like with our Turing machines, we fetch instructions from an abacus program such that a jump out of "range" behaves like a *Nop*-action. In this way it is again easy to define a function *steps* that executes n instructions of an abacus program. A *configuration* of an abacus machine is the current program counter together with a snapshot of all registers. By convention the value calculated by an abacus program is stored in the last register (the one with the highest index in the program).

The main point of abacus programs is to be able to translate them to Turing machine programs. Registers and their content are represented by standard tapes (see definition

shown in (3)). Because of the jumps in abacus programs, it is impossible to build Turing machine programs out of components using our $\oplus$-operation shown in the previous section. To overcome this difficulty, we calculate a *layout* of an abacus program as follows

$$\begin{array}{lcl} \mathit{layout}\ [] & \stackrel{\mathit{def}}{=} & [] \\ \mathit{layout}\ (\mathit{Inc}\ R{::}\mathit{is}) & \stackrel{\mathit{def}}{=} & 2 * R + 9{::}\mathit{layout}\ \mathit{is} \\ \mathit{layout}\ (\mathit{Dec}\ R\ l{::}\mathit{is}) & \stackrel{\mathit{def}}{=} & 2 * R + 16{::}\mathit{layout}\ \mathit{is} \\ \mathit{layout}\ (\mathit{Goto}\ l{::}\mathit{is}) & \stackrel{\mathit{def}}{=} & 1{::}\mathit{layout}\ \mathit{is} \end{array}$$

This gives us a list of natural numbers specifying how many states are needed to translate each abacus instruction. This information is needed in order to calculate the state where the Turing machine program of an abacus instruction starts. This can be defined as

$$\mathit{address}\ p\ n = \mathit{Suc}\ (\Sigma\ (\mathit{take}\ n\ (\mathit{layout}\ p)))$$

where p is an abacus program and *take* n takes the first n elements from a list.

The *Goto* instruction is easiest to translate requiring only one state, namely the Turing machine program:

$$\mathit{translate\_Goto}\ l \stackrel{\mathit{def}}{=} [(\mathit{Nop}, l), (\mathit{Nop}, l)]$$

where l is the state in the Turing machine program to jump to. For translating the instruction *Inc* R, one has to remember that the content of the registers are encoded in the Turing machine as a standard tape. Therefore the translated Turing machine needs to first find the number corresponding to the content of register R. This needs a machine with $2 * R$ states and can be constructed as follows:

$$\begin{array}{l} \mathit{TMFindnth}\ 0 \stackrel{\mathit{def}}{=} [] \\ \mathit{TMFindnth}\ (\mathit{Suc}\ n) \stackrel{\mathit{def}}{=} \\ \quad \mathit{TMFindnth}\ n\ @\ [(W_{Oc}, 2 * n + 1), (R, 2 * n + 2), (R, 2 * n + 3), (R, 2 * n + 2)] \end{array}$$

Then we need to increase the "number" on the tape by one, and adjust the following "registers". For adjusting we only need to change the first *Oc* of each number to *Bk* and the last one from *Bk* to *Oc*. Finally, we need to transition the head of the Turing machine back into the standard position. This requires a Turing machine with 9 states (we omit the details). Similarly for the translation of *Dec* R l, where the translated Turing machine needs to first check whether the content of the corresponding register is *0*. For this we have a Turing machine program with *16* states (again the details are omitted).

Finally, having a Turing machine for each abacus instruction we need to "stitch" the Turing machines together into one so that each Turing machine component transitions to next one, just like in the abacus programs. One last problem to overcome is that an abacus program is assumed to calculate a value stored in the last register (the one with the highest register). That means we have to append a Turing machine that "mops up" the tape (cleaning all *Oc*s) except for the *Oc*s of the last register represented on the tape.

This needs a Turing machine program with $2 * R + 6$ states, assuming R is the number of registers to be "cleaned".

While generating the Turing machine program for an abacus program is not too difficult to formalise, the problem is that it contains *Goto*s all over the place. The unfortunate result is that we cannot use our Hoare-rules for reasoning about sequentially composed programs (for this each component needs to be completely independent). Instead we have to treat the translated Turing machine as one "big block" and prove as invariant that it performs the same operations as the abacus program. For this we have to show that for each configuration of an abacus machine the *step*-function is simulated by zero or more steps in our translated Turing machine. This leads to a rather large "monolithic" correctness proof (4600 loc and 380 sublemmas) that on the conceptual level is difficult to break down into smaller components.

4 Recursive Functions and a Universal Turing Machine

The main point of recursive functions is that we can relatively easily construct a universal Turing machine via a universal function. This is different from Norrish [8] who gives a universal function for the lambda-calculus, and also from Asperti and Ricciotti [1] who construct a universal Turing machine directly, but for simulating Turing machines with a more restricted alphabet. Unlike Norrish [8], we need to represent recursive functions "deeply" because we want to translate them to abacus programs. Thus *recursive functions* are defined as the datatype

$$\begin{array}{lll} r ::= z & \text{(zero-function)} & \mid Cn^n\, f\, fs \quad \text{(composition)} \\ \mid\ s & \text{(successor-function)} & \mid Pr^n\, f_1\, f_2 \quad \text{(primitive recursion)} \\ \mid\ id^n_m & \text{(projection)} & \mid Mn^n\, f \quad \text{(minimisation)} \end{array}$$

where n indicates the function expects n arguments (in [2] both z and s expect one argument), and fs stands for a list of recursive functions. Since we know in each case the arity, say l, we can define an evaluation function, called *eval*, that takes a recursive function f and a list ns of natural numbers of length l as arguments. Since this evaluation function uses the minimisation operator from HOL, this function might not terminate always. As a result we also need to inductively characterise when *eval* terminates. We omit the definitions for *eval f ns* and *terminate f ns*. Because of space reasons, we also omit the definition of translating recursive functions into abacus programs. We can prove, however, the following theorem about the translation: If *rec_calc_rel f ns r* holds for the recursive function f and arguments ns, then the following Hoare-triple holds

$$\{\lambda tp.\ tp = ([Bk, Bk], \langle ns \rangle)\}\ translate\, f\ \{\lambda tp.\ \exists\, k\, l.\ tp = (Bk^k, \langle r \rangle\ @\ Bk^l)\}$$

for the Turing machine generated by *translate f*. This means the translated Turing machine if started with the standard tape $([Bk, Bk], \langle ns \rangle)$ will terminate with the standard tape $(Bk^k, \langle eval\, f\, ns \rangle\ @\ Bk^l)$ for some k and l.

Having recursive functions under our belt, we can construct a universal function, written *UF*. This universal function acts like an interpreter for Turing machines. It takes two arguments: one is the code of the Turing machine to be interpreted and the other

is the "packed version" of the arguments of the Turing machine. We can then consider how this universal function is translated to a Turing machine and from this construct the universal Turing machine, written *UTM*. *UTM* is defined as the composition of the Turing machine that packages the arguments and the translated recursive function *UF*:

$$UTM \stackrel{def}{=} arg\_coding \oplus (translate\ UF)$$

Suppose a Turing program p is well-formed and when started with the standard tape containing the arguments *args*, will produce a standard tape with "output" n. This assumption can be written as the Hoare-triple

$$\{\lambda tp.\ tp = ([], \langle args\rangle)\}\ p\ \{\lambda tp.\ tp = (Bk^m, \langle n\rangle\ @\ Bk^k)\}$$

where we require that the *args* stand for a non-empty list. Then the universal Turing machine *UTM* started with the code of p and the arguments *args*, calculates the same result, namely

$$\{\lambda tp.\ tp = ([], \langle code\ p::args\rangle)\}\ UTM\ \{\lambda tp.\ \exists m\ k.\ tp = (Bk^m, \langle n\rangle\ @\ Bk^k)\}$$

Similarly, if a Turing program p started with the standard tape containing *args* loops, which is represented by the Hoare-pair

$$\{\lambda tp.\ tp = ([], \langle args\rangle)\}\ p \uparrow$$

then the universal Turing machine started with the code of p and the arguments *args* will also loop

$$\{\lambda tp.\ tp = ([], \langle code\ p::args\rangle)\}\ UTM \uparrow$$

While formalising the chapter in [2] about universal Turing machines, an unexpected outcome of our work is that we identified an inconsistency in their use of a definition. This is unexpected since [2] is a classic textbook which has undergone several editions (we used the fifth edition; the material containing the inconsistency was introduced in the fourth edition of this book). The central idea about Turing machines is that when started with standard tapes they compute a partial arithmetic function. The inconsistency arises when they define the case when this function should *not* return a result. Boolos at al write in Chapter 3, Page 32:

> *"If the function that is to be computed assigns no value to the arguments that are represented initially on the tape, then the machine either will never halt, or will halt in some nonstandard configuration..."*

Interestingly, they do not implement this definition when constructing their universal Turing machine. In Chapter 8, on page 93, a recursive function *stdh* is defined as:

$$stdh(m, x, t) \stackrel{def}{=} stat(conf(m, x, t)) + nstd(conf(m, x, t)) \tag{6}$$

where $stat(conf(m, x, t))$ computes the current state of the simulated Turing machine, and $nstd(conf(m, x, t))$ returns *1* if the tape content is non-standard. If either one evaluates to something that is not zero, then $stdh(m, x, t)$ will be not zero, because of the

+-operation. On the same page, a function $halt(m, x)$ is defined in terms of *stdh* for computing the steps the Turing machine needs to execute before it halts (in case it halts at all). According to this definition, the simulated Turing machine will continue to run after entering the *0*-state with a non-standard tape. The consequence of this inconsistency is that there exist Turing machines that given some arguments do not compute a value according to Chapter 3, but return a proper result according to the definition in Chapter 8. One such Turing machine is:

$$counter\_example \stackrel{def}{=} [(L, 0), (L, 2), (R, 2), (R, 0)]$$

If started with standard tape $([], [Oc])$, it halts with the non-standard tape $([Oc, Bk], [])$ according to the definition in Chapter 3—so no result is calculated; but with the standard tape $([], [Oc])$ according to the definition in Chapter 8. ???? We solve this inconsistency in our formalisation by setting up our definitions so that the *counter_example* Turing machine does not produce any result by looping forever fetching *Nop*s in state *0*. This solution implements essentially the definition in Chapter 3; it differs from the definition in Chapter 8, where perplexingly the instruction from state *1* is fetched.

5 Conclusion

In previous works we were unable to formalise results about computability because in Isabelle/HOL we cannot represent the decidability of a predicate P, say, as the formula $P \vee \neg P$. For reasoning about computability we need to formalise a concrete model of computations. We could have followed Norrish [8] using the λ-calculus as the starting point for computability theory, but then we would still have to connect his work to Turing machines for proofs that make essential use of them (for example the undecidability proof for Wang's tiling problem [10]).

We therefore have formalised Turing machines in the first place and the main computability results from Chapters 3 to 8 in the textbook by Boolos et al [2]. For this we did not need to implement anything on the ML-level of Isabelle/HOL. While formalising the six chapters of [2] we have found an inconsistency in Boolos et al's definitions of what function a Turing machine calculates. In Chapter 3 they use a definition that states a function is undefined if the Turing machine loops *or* halts with a non-standard tape. Whereas in Chapter 8 about the universal Turing machine, the Turing machines will *not* halt unless the tape is in standard form. If the title had not already been taken in [7], we could have titled our paper "Boolos et al are (almost) Right". We have not attempted to formalise everything precisely as Boolos et al present it, but use definitions that make our mechanised proofs manageable. For example our definition of the halting state performing *Nop*-operations seems to be non-standard, but very much suited to a formalisation in a theorem prover where the *steps*-function needs to be total.

Norrish mentions that formalising Turing machines would be a "*daunting prospect*" [8, Page 310]. While λ-terms indeed lead to some slick mechanised proofs, our experience is that Turing machines are not too daunting if one is only concerned with formalising the undecidability of the halting problem for Turing machines. As a point of comparison, the halting problem took us around 1500 loc of Isar-proofs, which is just

one-and-a-half times of a mechanised proof pearl about the Myhill-Nerode theorem. So our conclusion is that this part is not as daunting as we estimated when reading the paper by Norrish [8]. The work involved with constructing a universal Turing machine via recursive functions and abacus machines, we agree, is not a project one wants to undertake too many times (our formalisation of abacus machines and their correct translation is approximately 4600 loc; recursive functions 2800 loc and the universal Turing machine 10000 loc).

Our work is also very much inspired by the formalisation of Turing machines of Asperti and Ricciotti [1] in the Matita theorem prover. It turns out that their notion of realisability and our Hoare-triples are very similar, however we differ in some basic definitions for Turing machines. Asperti and Ricciotti are interested in providing a mechanised foundation for complexity theory. They formalised a universal Turing machine (which differs from ours by using a more general alphabet), but did not describe an undecidability proof. Given their definitions and infrastructure, we expect however this should not be too difficult for them.

For us the most interesting aspects of our work are the correctness proofs for Turing machines. Informal presentations of computability theory often leave the constructions of particular Turing machines as exercise to the reader, for example [2], deeming it to be just a chore. However, as far as we are aware all informal presentations leave out any arguments why these Turing machines should be correct. This means the reader is left with the task of finding appropriate invariants and measures for showing the correctness and termination of these Turing machines. Whenever we can use Hoare-style reasoning, the invariants are relatively straightforward and again as a point of comparison much smaller than for example the invariants used by Myreen in a correctness proof of a garbage collector written in machine code [6, Page 76]. However, the invariant needed for the abacus proof, where Hoare-style reasoning does not work, is similar in size as the one by Myreen and finding a sufficiently strong one took us, like Myreen, something on the magnitude of weeks.

Our reasoning about the invariants is not much supported by the automation beyond the standard automation tools available in Isabelle/HOL. There is however a tantalising connection between our work and very recent work by Jensen et al [4] on verifying X86 assembly code that might change that. They observed a similar phenomenon with assembly programs where Hoare-style reasoning is sometimes possible, but sometimes it is not. In order to ease their reasoning, they introduced a more primitive specification logic, on which Hoare-rules can be provided for special cases. It remains to be seen whether their specification logic for assembly code can make it easier to reason about our Turing programs. Myreen ??? That would be an attractive result, because Turing machine programs are very much like assembly programs and it would connect some very classic work on Turing machines to very cutting-edge work on machine code verification. In order to try out such ideas, our formalisation provides the "playground". The code of our formalisation is available from the Mercurial repository at http://www.dcs.kcl.ac.uk/staff/urbanc/cgi-bin/repos.cgi/tm/.

Acknowledgements: We are very grateful for the extremely helpful comments by the anonymous reviewers.

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. J. B. Jensen, N. Benton, and A. Kennedy. High-Level Separation Logic for Low-Level Code. In *Proc. of the 40th Symposium on Principles of Programming Languages (POPL)*, pages 301–314, 2013.
5. A. Krauss. Recursive Definitions of Monadic Functions. In *Proc. of the Workshop on Partiality and Recursion in Interactive Theorem Provers*, volume 43 of *EPTCS*, pages 1–13, 2010.
6. M. O. Myreen. *Formal Verification of Machine-Code Programs*. PhD thesis, University of Cambridge, 2009.
7. T. Nipkow. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
8. M. Norrish. Mechanised Computability Theory. In *Proc. of the 2nd Conference on Interactive Theorem Proving (ITP)*, volume 6898 of *LNCS*, pages 297–311, 2011.
9. E. Post. Finite Combinatory Processes-Formulation 1. *Journal of Symbolic Logic*, 1(3):103–105, 1936.
10. R. M. Robinson. Undecidability and Nonperiodicity for Tilings of the Plane. *Inventiones Mathematicae*, 12:177–209, 1971.
11. V. Zammit. *On the Readability of Machine Checkable Formal Proofs*. PhD thesis, University of Kent, 1999.