IMP is Turing-Complete

Tassilo Lemke

January 15, 2025

Abstract

In this project we show the Turing-Completeness of the "simple imperative programming language" IMP, as described in "Concrete Semantics" by Tobias Nipkow and Gerwin Klein. We do this by showing, that every Turing-Machine can be simulated by an IMP-program.

Contents

1 Introduction

1.1 Background

When investigating different models of computation, we often are interested in their expressiveness and powerfulness, that is determining which type of computations can be done. By a long-standing claim of Church and Turing we, to this day, believe that the most powerful machines we can have, are those that can perform every a computation a Turing-Machine can compute as well. [1] The most interesting question we can thus pose about a computational model, is whether it is *Turing-Complete* or not. That is, is the model capable of computing everything a Turing-Machine can compute. The most common approach is to either show, that a model can simulate any Turing-Machine, or that it can simulate a specific Turing-Machine: the Universal Turing Machine. [4]

1.2 Motivation

IMP is a "minimalistic imperative programming language" introduced by Nipkow and Klein in their works of formalising programming language semantics. [2] Altough it is claimed to be Turing-Complete, a formal proof of this fact is not of my knowledge. Therefore, in this project, we attempt to fully prove the Turing-Completeness of IMP.

1.3 Proof Sketch

Since the definition of Turing-Machines we use only requires two state symbols (*Bk* and *Oc*), we can make use of a binary representation to encode tapes, where 0 represents the blank symbol *Bk*, while 1 represents the other symbol *Oc*. That two state symbols are sufficient to simulate any Turing-Machine, given enough states, has already been shown by Claude E. Shannon in 1956.[3]

This is also beneficial, as it allows us to represent an infinite tape, without having to deal with actual infinite data structures, since a natural number has an infinite amount of zeroes in its binary representation in theory. Pushing and reading from the tape can then be achieved using only multiplication by two, integer-division by two and its remainder.

We will later construct an IMP-program that stores the infinite tapes to the left and right in two variables, and uses two more variables to store the current state and head. The IMP-program will then repeat until it reaches a final state, executing the next transition of the Turing-Machine with each step.

We can then show that when the IMP-program terminates, its variables represent the same configuration the "normal" Turing-Machine would have when it would terminate.

1.4 Dependencies

We use the Hoare Logic for both Partial and Total Correctness of IMP to prove our constructed program correct. Furthermore, we take an existing definition of Turing-Machines from the AFP[5], although we will create a slightly different intermediate definition later on.

theory *IMP-TuringComplete*
  imports
    HOL−IMP.Hoare-Total

HOL−IMP.Hoare-Sound-Complete
Universal-Turing-Machine.Turing-aux
begin — begin-theory IMP_TuringComplete

During the time of writing, the Hoare Logic was the most powerful and latest introduced tool for reasoning about semantics of my knowledge. There are surely ways to make some proofs shorter and more elegant. However, I took it as a challenge and exercise to use a Hoare Logic and only it, meaning also no Verification Condition Generation.

2 Intermediate Representation

The imported definition of Turing-Machines defines a tape-configuration slightly differently than usual, deviating from the standard 3-tuple (L, H, R), containing the infinite tape to the left and right and the current head symbol, by only using the 2-tuple (L, R), where the head symbol is the first symbol on the right tape.

While this definition may seem to make next to no difference, we will save us much trouble by defining an intermediate representation using the standard 3-tuple right away. Furthermore, we will directly encode the infinite tapes to the left and right as natural numbers, which will make our definition both more well-defined, see section 2.1.1, and make some later proofs easier.

2.1 Tape Translation

2.1.1 Defining Tape Equivalence

The definition of Turing-Machines in the "Universal Turing Machine"[5] project allows for inherently ambiguous tapes (namely having trailing blanks). Since the tapes have infinite trailing blanks in theory, this effectively means that the standard = relation is insufficient.

Because of this, we define a custom equality relation $=_T$, which specifies the equality of the tape contents. To make things easier for us later, we also define a custom equality relation $=_C$, which specifies the equality of an entire TM-configuration.

fun *tape-eq* :: *cell list* ⇒ *cell list* ⇒ *bool* (**infix** $=_T$ *55*) **where**
$((x\#xs) =_T (y\#ys)) = ((x = y) \wedge (xs =_T ys))$ |
$((x\#xs) =_T []) = ((x = Bk) \wedge (xs =_T []))$ |
$([] =_T (y\#ys)) = ((y = Bk) \wedge ([] =_T ys))$ |
$([] =_T []) = True$

fun *config-eq* :: *config* ⇒ *config* ⇒ *bool* (**infix** $=_C$ *55*) **where**
$((s1, l1, r1) =_C (s2, l2, r2)) = ((s1 = s2) \wedge (l1 =_T l2) \wedge (r1 =_T r2))$

lemma *tape-eq-correct*:
assumes $xs =_T ys$
shows *read xs* = *read ys* **and** *tl xs* $=_T$ *tl ys*
using *assms* **by** (*induction xs ys rule*: *tape-eq.induct*) *simp+*

Future Note: Given the latest lecture I am now convinced a much more elegant solution for this would have been to use equivalence classes and/or a quotient datatype. Sadly, during the time of writing I was not aware of this feature, which now introduces this rather complicated custom equivalence relation.

We now prove some generic properties of the equality relation, namely reflexivity, symmetry and transitivity.

lemma *tape-eq-refl*: $xs =_T xs$
  by (*induction xs*) *simp+*

lemma *config-eq-refl*: $(s, l, r) =_C (s, l, r)$
  using *tape-eq-refl* **by** *simp*

lemma *config-eq-refl'*: $c =_C c$
  using *prod-cases3 config-eq-refl* **by** *metis*

lemma *tape-eq-sym*: $xs =_T ys \Longrightarrow ys =_T xs$
  by (*induction xs ys rule*: *tape-eq.induct*) *simp+*

lemma *config-eq-sym*: $(s1, l1, r1) =_C (s2, l2, r2) \Longrightarrow (s2, l2, r2) =_C (s1, l1, r1)$
  using *tape-eq-sym* **by** *simp*

lemma *config-eq-sym'*: $c1 =_C c2 \Longrightarrow c2 =_C c1$
  using *prod-cases3 config-eq-sym* **by** *metis*

lemma *tape-eq-trans*: $xs =_T ys \Longrightarrow ys =_T zs \Longrightarrow xs =_T zs$
proof (*induction xs zs arbitrary*: *ys rule*: *tape-eq.induct*)
  case (*1 x xs z zs*)
  then show *?case* **by** (*induction ys*) *force+*
next
  case (*2 x xs*)
  then show *?case* **by** (*induction ys*) *force+*
next
  case (*3 z zs*)
  then show *?case* **by** (*induction ys*) *force+*
next
  case *4*
  then show *?case* **by** *simp*
qed

lemma *config-eq-trans*:
  assumes $(s1, l1, r1) =_C (s2, l2, r2)$
    and $(s2, l2, r2) =_C (s3, l3, r3)$
  shows $(s1, l1, r1) =_C (s3, l3, r3)$
  using *assms tape-eq-trans*[*of l1 l2 l3*] *tape-eq-trans*[*of r1 r2 r3*] **by** *simp*

lemma *config-eq-trans'*: $c1 =_C c2 \Longrightarrow c2 =_C c3 \Longrightarrow c1 =_C c3$
  using *prod-cases3 config-eq-trans* **by** *metis*

2.1.2 Translation: Tape $\Longleftrightarrow$ Natural Numbers

We want to use natural numbers, specifically their binary representation, to encode tapes. This has the benefit of uniquely identifying tapes, since theoretically the binary representations also have an infinite number of zeroes (the blank symbol).

Using the binary representation means shifting the tape is as simple as multiplying or dividing by 2, to extract the top element we can use modulo 2.

fun *cell-to-nat* :: *cell* ⇒ *nat* **where**
  cell-to-nat Bk = *0* |
  cell-to-nat Oc = *1*

fun *nat-to-cell* :: *nat* ⇒ *cell* **where**
  nat-to-cell 0 = *Bk* |
  nat-to-cell n = *Oc*

fun *tape-to-nat* :: *cell list* ⇒ *nat* **where**
  tape-to-nat (*x*#*xs*) = *2* ∗ *tape-to-nat xs* + (*cell-to-nat x*) |
  tape-to-nat [] = *0*

lemma *tape-to-nat-det*:
  assumes $xs =_T ys$
  shows *tape-to-nat xs* = *tape-to-nat ys*
  using *assms* **by** (*induction xs ys rule*: *tape-eq.induct*) *simp+*

fun *nat-to-tape* :: *nat* ⇒ *cell list* **where**
  nat-to-tape 0 = [] |
  nat-to-tape n = (*nat-to-cell* (*n mod 2*))#(*nat-to-tape* (*n div 2*))

2.1.3 Tape Operations

We can now prove that multiplication, division and modulo behave as expected.

lemma *mul2-is-push-bk*:
  assumes *nat-to-tape n* $=_T$ *xs*
  shows *nat-to-tape* (*2*∗*n*) $=_T$ (*Bk*#*xs*)
  using *assms* **by** (*cases n*) *simp+*

lemma *mul2-is-push-oc*:
  assumes *nat-to-tape n* $=_T$ *xs*
  shows *nat-to-tape* (*2*∗*n* + *1*) $=_T$ (*Oc*#*xs*)
  using *assms* **by** *simp*

lemma *mul2-is-push*:
  assumes *nat-to-tape n* $=_T$ *xs*
  shows *nat-to-tape* (*2*∗*n* + *cell-to-nat x*) $=_T$ (*x*#*xs*)
proof (*cases x*)
  case *Bk* **thus** *?thesis* **by** (*simp add*: *assms mul2-is-push-bk*)
next
  case *Oc* **thus** *?thesis* **by** (*simp add*: *assms mul2-is-push-oc*)
qed

lemma *div2-is-pop*:
  assumes *nat-to-tape n* $=_T$ *xs*
  shows *nat-to-tape* (*n div 2*) $=_T$ *tl xs*
proof (*cases n* ≠ *0*)

```
  case True
  then have nat-to-tape n ≠ []
    using nat-to-tape.elims by blast
  then obtain y ys where ys-def: (y#ys) = nat-to-tape n
    by (metis nat-to-tape.elims)
  then have ys =_T tl xs
    by (metis assms list.collapse list.sel(2) tape-eq.simps(1) tape-eq.simps(2))
  moreover have ys = nat-to-tape (n div 2)
    by (metis ys-def True list.inject nat-to-tape.elims)
  ultimately show ?thesis
    by simp
next
  case False
  with assms show ?thesis by (cases xs) (use tape-eq.simps in simp)+
qed

lemma mod2-is-read:
  assumes nat-to-tape n =_T xs
  shows nat-to-cell (n mod 2) = read xs
proof (cases n ≠ 0)
  case True
  then have nat-to-tape n ≠ []
    using nat-to-tape.elims by blast
  then obtain y ys where ys-def: (y#ys) = nat-to-tape n
    by (metis nat-to-tape.elims)
  then have read xs = y
  proof (cases xs)
    case Nil
    then have (y#ys) =_T []
      using assms ys-def by simp
    then show ?thesis
      using tape-eq.simps(2) Nil by simp
  next
    case (Cons x xs)
    then have (y#ys) =_T (x#xs)
      using assms ys-def by simp
    then show ?thesis
      using tape-eq.simps(1)[of y ys x xs] Cons by simp
  qed
  moreover have nat-to-cell (n mod 2) = y
    by (metis ys-def True list.inject nat-to-tape.elims)
  ultimately show ?thesis by simp
next
  case False
  with assms show ?thesis by (cases xs) (use tape-eq.simps in simp)+
qed

lemma nat-to-cell-ident: n < 2 ⟹ cell-to-nat (nat-to-cell n) = n
  by (induction n rule: nat-to-cell.induct) simp+
```

lemma *cell-to-nat-ident*: *nat-to-cell* (*cell-to-nat c*) = *c*
  by (*cases c*) *simp*+

lemma *nat-to-tape-ident*: *tape-to-nat* (*nat-to-tape n*) = *n*
  by (*induction n rule*: *nat-to-tape.induct*) (*use nat-to-cell-ident* **in** *simp*)+

lemma *tape-to-nat-ident*: *nat-to-tape* (*tape-to-nat xs*) $=_T$ *xs*
proof (*induction xs rule*: *tape-to-nat.induct*)
  case (*1 x xs*)
  then show *?case* **by** (*cases x*) (*simp add*: *mul2-is-push-bk*)+
next
  case *2*
  then show *?case* **by** *simp*
qed

lemma *tape-to-nat-ident-read*: *nat-to-cell* ((*tape-to-nat xs*) *mod 2*) = *read xs*
proof (*cases xs*)
  case *Nil*
  then show *?thesis* **by** *simp*
next
  case (*Cons x xs*)
  then show *?thesis* **by** (*cases x*) *simp*+
qed

lemma *tape-to-nat-ident-tl*: *nat-to-tape* ((*tape-to-nat xs*) *div 2*) $=_T$ *tl xs*
proof (*cases xs*)
  case *Nil*
  then show *?thesis* **by** *simp*
next
  case (*Cons x xs*)
  then show *?thesis* **by** (*cases x*) (*use tape-to-nat-ident* **in** *simp*)+
qed

lemma *tape-eq-merge*: $r' =_T$ *tl r* $\Longrightarrow$ *read r* = *h* $\Longrightarrow$ $h \# r' =_T r$
  by (*cases r*) *simp*+

lemma *tape-eq-merge2*: $r \neq []$ $\Longrightarrow$ *hd r* # *nat-to-tape* (*tape-to-nat* (*tl r*)) $=_T r$
  by (*simp add*: *tape-eq-merge tape-to-nat-ident*)

2.2 Intermediate State

Our intermediate-tape is a 3-tuple (L, H, R), where:

- L is the infinite tape to the left, represented as a natural number
- R is the inifnite tape to the right, represented as a natural number
- H is the symbol at the current head

type-synonym *itape* = *nat* × *cell* × *nat*

The complete intermediate-state is then a 4-tuple (S, L, H, R), where:

- S is the current state of the TM
- (L, H, R) is the intermediate-tape, as described above

type-synonym *istate* = *state* × *itape*

fun *config-to-istate* :: *config* ⇒ *istate* **where**
  config-to-istate (*s*, *l*, *r*) = (*s*, *tape-to-nat l*, *read r*, *tape-to-nat* (*tl r*))

lemma *config-to-istate-det*:
  assumes (*s1*, *l1*, *r1*) $=_C$ (*s2*, *l2*, *r2*)
  shows *config-to-istate* (*s1*, *l1*, *r1*) = *config-to-istate* (*s2*, *l2*, *r2*)
proof −
  let *?s1* = (*s1*, *l1*, *r1*)
  let *?s2* = (*s2*, *l2*, *r2*)

  obtain *s1' l1' h1' r1'* **where** *st1-def*: (*s1'*, *l1'*, *h1'*, *r1'*) = *config-to-istate ?s1*
    by *simp*
  obtain *s2' l2' h2' r2'* **where** *st2-def*: (*s2'*, *l2'*, *h2'*, *r2'*) = *config-to-istate ?s2*
    by *simp*

  have *s1* = *s2*
    using *assms* **by** *simp*
  then have *s-eq*: *s1'* = *s2'*
    using *st1-def st2-def* **by** *simp*

  have *l1* $=_T$ *l2*
    using *assms* **by** *simp*
  then have *l-eq*: *l1'* = *l2'*
    using *st1-def st2-def* **by** (*simp add*: *tape-to-nat-det*)

  have *r1* $=_T$ *r2*
    using *assms* **by** *simp*
  moreover have *tl r1* $=_T$ *tl r2*
    using *tape-eq-correct calculation* **by** *simp*
  ultimately have *h-eq*: *h1'* = *h2'* **and** *r-eq*: *r1'* = *r2'*
    using *tape-eq-correct tape-to-nat-det st1-def st2-def* **by** *simp+*

  from *s-eq l-eq h-eq r-eq* **show** *?thesis*
    using *st1-def st2-def* **by** *simp*
qed

lemma *config-to-istate-det'*:
  assumes *c1* $=_C$ *c2*
  shows *config-to-istate c1* = *config-to-istate c2*
  using *assms prod-cases3 config-to-istate-det* **by** *metis*

fun *istate-to-config* :: *istate* ⇒ *config* **where**

istate-to-config (*s*, *l*, *h*, *r*) = (*s*, *nat-to-tape l*, *h*#(*nat-to-tape r*))

lemma *config-to-istate-ident*: *istate-to-config* (*config-to-istate* (*s*, *l*, *r*)) $=_C$ (*s*, *l*, *r*)
by (*simp add*: *tape-to-nat-ident tape-eq-merge2*)

lemma *config-to-istate-ident'*: *istate-to-config* (*config-to-istate c*) $=_C$ *c*
using *prod-cases3 config-to-istate-ident* **by** *metis*

lemma *istate-to-config-ident*: *config-to-istate* (*istate-to-config* (*s*, *l*, *h*, *r*)) = (*s*, *l*, *h*, *r*)
by (*simp add*: *nat-to-tape-ident*)

lemma *istate-to-config-ident'*: *config-to-istate* (*istate-to-config st*) = *st*
using *prod-cases3 istate-to-config-ident* **by** *metis*

lemma *istate-to-config-l*: *istate-to-config* (*s*, *l*, *h*, *r*) = (*s'*, *l'*, *r'*) $\Longrightarrow$ *nat-to-tape l* $=_T$ *l'*
using *config-eq-refl* **by** *simp*

lemma *istate-to-config-r*: *istate-to-config* (*s*, *l*, *h*, *r*) = (*s'*, *l'*, *r'*) $\Longrightarrow$ *nat-to-tape r* $=_T$ *tl r'*
using *config-eq-refl* **by** (*cases r'*) *simp+*

lemma *istate-to-config-h*: *istate-to-config* (*s*, *l*, *h*, *r*) = (*s'*, *l'*, *r'*) $\Longrightarrow$ *h* = *read r'*
by (*cases r'*) *simp+*

lemma *istate-to-config-s*: *istate-to-config* (*s*, *l*, *h*, *r*) = (*s'*, *l'*, *r'*) $\Longrightarrow$ *s* = *s'*
by *simp*

2.3 Single-Step Execution

fun *iupdate* :: *instr* ⇒ *istate* ⇒ *istate* **where**
iupdate (*WB*, *s'*) (*s*, *l*, *h*, *r*) = (*s'*, *l*, *Bk*, *r*) |
iupdate (*WO*, *s'*) (*s*, *l*, *h*, *r*) = (*s'*, *l*, *Oc*, *r*) |
iupdate (*L*, *s'*) (*s*, *l*, *h*, *r*) = (*s'*, *l div 2*, *nat-to-cell* (*l mod 2*), *2*∗*r* + *cell-to-nat h*) |
iupdate (*R*, *s'*) (*s*, *l*, *h*, *r*) = (*s'*, *2*∗*l* + *cell-to-nat h*, *nat-to-cell* (*r mod 2*), *r div 2*) |
iupdate (*Nop*, *s'*) (*s*, *l*, *h*, *r*) = (*s'*, *l*, *h*, *r*)

lemma *iupdate-correct*:
assumes (*s'*, *l'*, *h'*, *r'*) = *config-to-istate* (*s*, *l*, *r*)
shows *istate-to-config* (*iupdate* (*a*, *s''*) (*s'*, *l'*, *h'*, *r'*)) $=_C$ (*s''*, *update a* (*l*, *r*))
proof (*cases a*)
case *WB*
then show *?thesis* **by** (*simp add*: *assms tape-to-nat-ident*)
next
case *WO*
then show *?thesis* **by** (*simp add*: *assms tape-to-nat-ident*)
next
case *L*
then have *iupdate* (*a*, *s''*) (*s'*, *l'*, *h'*, *r'*) = (*s''*, *l' div 2*, *nat-to-cell* (*l' mod 2*), *2*∗*r'* + *cell-to-nat h'*)
by *simp*
moreover have (*s''*, *update a* (*l*, *r*)) = (*s''*, *tl l*, (*read l*)#*r*)

using L **by** (*cases l*) *simp+*
moreover have *nat-to-tape* (l' *div 2*) $=_T$ *tl l*
using *assms* **by** (*simp add*: *tape-to-nat-ident div2-is-pop*)
moreover have *nat-to-tape* ($2{*}r'$ + *cell-to-nat* h') $=_T$ $h'\#(tl\ r)$
using *assms* **by** (*simp add*: *tape-to-nat-ident mul2-is-push*)
moreover have $h'\#(tl\ r) =_T r$
using *assms* **by** (*simp add*: *tape-eq-refl*)
moreover have *nat-to-tape* ($2{*}r'$ + *cell-to-nat* h') $=_T$ r
using *calculation*(*4*) *calculation*(*5*) *tape-eq-trans* **by** *blast*
ultimately show *?thesis*
using *assms* **by** (*simp add*: *mod2-is-read tape-to-nat-ident*)
next
case R
then have *iupdate* (a, s'') (s', l', h', r') = (s'', $2{*}l'$ + *cell-to-nat* h', *nat-to-cell* (r' *mod 2*), r' *div 2*)
by *simp*
moreover have (s'', *update a* (l, r)) = (s'', (*read r*)$\#l$, *tl r*)
using R **by** (*cases r*) *simp+*
moreover have *nat-to-tape* (r' *div 2*) $=_T$ *tl* (*tl r*)
using *assms* **by** (*simp add*: *tape-to-nat-ident div2-is-pop*)
moreover have *nat-to-tape* ($2{*}l'$ + *cell-to-nat* h') $=_T$ $h'\#l$
using *assms* **by** (*simp add*: *tape-to-nat-ident mul2-is-push*)
moreover have $h'\#l =_T$ (*read r*)$\#l$
using *assms* **by** (*simp add*: *tape-eq-refl*)
moreover have *nat-to-tape* ($2{*}l'$ + *cell-to-nat* h') $=_T$ (*read r*)$\#l$
using *calculation*(*4*) *calculation*(*5*) *tape-eq-trans* **by** *blast*
moreover have (*nat-to-cell* (r' *mod 2*))$\#$*nat-to-tape* (r' *div 2*) $=_T$ *tl r*
using *calculation*(*3*) *assms tape-to-nat-ident-read tape-to-nat-ident-tl tape-eq-merge* **by** *simp*
ultimately show *?thesis*
using *assms* **by** *simp*
next
case *Nop*
then show *?thesis* **by** (*simp add*: *assms tape-to-nat-ident tape-eq-merge*)
qed

lemma *config-eq-det-is-final*:
assumes (*s1*, *l1*, *r1*) $=_C$ (*s2*, *l2*, *r2*)
shows *is-final* (*s1*, *l1*, *r1*) = *is-final* (*s2*, *l2*, *r2*)
using *assms* **by** *simp*

lemma *config-eq-det-is-final'*: *c1* $=_C$ *c2* $\Longrightarrow$ *is-final c1* = *is-final c2*
using *config-eq-det-is-final prod-cases3* **by** *metis*

2.4 Correct-Step Execution

2.4.1 Instruction-Index

The UTM project we use as dependency executes step, by calculating an instruction index. Turing-Machines are represented as lists of instructions and the corresponding instruction is then executed.

The original calculation of this index is:

$$i := 2s + h$$

where s is the current state and h is the current head symbol (0 for Bk, 1 for Oc).

fun *istate-to-index* :: *istate* ⇒ *nat* **where**
  istate-to-index (*s*, *l*, *h*, *r*) = *2∗s* + *cell-to-nat h*

abbreviation *config-to-index* :: *config* ⇒ *nat* **where**
  config-to-index c ≡ *istate-to-index* (*config-to-istate c*)

When executing the corresponding instruction, the UTM project first checks if we are in a final state, and would short-circuit to execute a *Nop* instruction. However, we avoid this short-circuiting by calculating the real instruction index. If it were a final state ($s = 0$), the resulting index would be 0 or 1. Thereby, we simply "prepend" two *Nop* instructions to the Turing-Machine. The actual implementation is a bit different and more ugly, by actually checking the value, however the behavior is semantically identical.

abbreviation *load-instr* :: *tprog0* ⇒ *nat* ⇒ *instr* (**infix** $@_I$ *55*) **where**
  tm $@_I$ *i* ≡ (*if i* < (*2* + *length tm*) ∧ *i* ≥ *2 then tm*!(*i*−*2*) *else* (*Nop*, *0*))

lemma *load-instr-correct*:
  tm $@_I$ (*istate-to-index* (*s*, *l*, *h*, *r*)) = *fetch tm s h*
proof (*induction tm s h rule*: *fetch.induct*)
  case (*1 p b*)
  then show *?case* **by** (*cases b*) *simp+*
next
  case (*2 p s*)
  then show *?case* **by** *simp*
next
  case (*3 p s*)
  then show *?case* **by** *simp*
qed

2.4.2 Instruction Execution

We now define step functions on our intermediate-state. Furthermore, we can prove that the instruction-index as explained above works as epxected.

fun *istep* :: *tprog0* ⇒ *istate* ⇒ *istate* **where**
  istep tm (*s*, *l*, *h*, *r*) = *iupdate* (*fetch tm s h*) (*s*, *l*, *h*, *r*)

lemma *istep-eq*:
  istep tm (*s*, *l*, *h*, *r*) = *iupdate* (*tm* $@_I$ (*istate-to-index* (*s*, *l*, *h*, *r*))) (*s*, *l*, *h*, *r*)
  using *load-instr-correct* **by** *simp*

lemma *istep-index-skip*:
  assumes *istate-to-index* (*s*, *l*, *h*, *r*) ≥ *2* + (*length tm*)
  shows *istep tm* (*s*, *l*, *h*, *r*) = (*0*, *l*, *h*, *r*)
  using *assms istep-eq* **by** *simp*

lemma *istep-index-skip'*:
  assumes *istate-to-index st* $\geq$ *2* + (*length tm*)
  shows *istep tm st* = *iupdate* (*Nop*, *0*) *st*
  using *assms istep-index-skip prod-cases4 iupdate.simps*(*5*) **by** *metis*

lemma *istep-index-correct*:
  assumes *istate-to-index* (*s*, *l*, *h*, *r*) < *2* + (*length tm*)
  shows *istep tm* (*s*, *l*, *h*, *r*) = *iupdate* (*tm* $@_I$ (*istate-to-index* (*s*, *l*, *h*, *r*))) (*s*, *l*, *h*, *r*)
  using *assms istep-eq* **by** *simp*

lemma *istep-index-correct'*:
  assumes *istate-to-index st* < *2* + (*length tm*)
  shows *istep tm st* = *iupdate* (*tm* $@_I$ (*istate-to-index st*)) *st*
proof −
  obtain *s l h r* **where** *st* = (*s*, *l*, *h*, *r*)
    using *prod-cases4* **by** *blast*
  then show *?thesis*
    using *assms istep-index-correct* **by** *simp*
qed

lemma *istep-correct*:
  assumes (*s1*, *l1*, *r1*) $\models\langle tm\rangle=$ (*s2*, *l2*, *r2*)
  shows *istep tm* (*config-to-istate* (*s1*, *l1*, *r1*)) = *config-to-istate* (*s2*, *l2*, *r2*)
proof −
  let *?s1* = (*s1*, *l1*, *r1*)
  let *?s2* = (*s2*, *l2*, *r2*)

  obtain *s1' l1' h1' r1'* **where** *st1-def*: (*s1'*, *l1'*, *h1'*, *r1'*) = *config-to-istate ?s1*
    by *simp*
  obtain *s2' l2' h2' r2'* **where** *st2-def*: (*s2'*, *l2'*, *h2'*, *r2'*) = *config-to-istate ?s2*
    by *simp*

  let *?st1* = (*s1'*, *l1'*, *h1'*, *r1'*)
  let *?st2* = (*s2'*, *l2'*, *h2'*, *r2'*)

  obtain *i1* **where** *i1-def*: *i1* = *fetch tm s1* (*read r1*)
    by *simp*
  obtain *i2* **where** *i2-def*: *i2* = *tm* $@_I$ *istate-to-index ?st1*
    by *simp*
  have *read r1* = *h1'* **and** *s1* = *s1'*
    using *st1-def* **by** *simp+*
  with *i1-def i2-def* **have** *i1* = *i2*
    using *load-instr-correct* **by** *simp*

  obtain *a s'* **where** *i1* = (*a*, *s'*) **and** *i2* = (*a*, *s'*)
    using ‹*i1* = *i2*› **by** *fastforce*

  have *step0 ?s1 tm* = *?s2*
    using *assms* **by** (*simp add*: *tm-step0-rel-def*)

then have *a1*: *?s2* = (*s*′, *update a* (*l1*, *r1*))
  using *i1-def* ‹*i1* = (*a*, *s*′)› **by** *force*

have *a2*: *istep tm ?st1* = *iupdate i2 ?st1*
  using *i2-def istep-eq* **by** *simp*

have *istate-to-config* (*iupdate i2 ?st1*) $=_C$ (*s*′, *update a* (*l1*, *r1*))
  using *iupdate-correct st1-def* ‹*i1* = *i2*› ‹*i2* = (*a*, *s*′)› **by** *simp*
then have *istate-to-config* (*istep tm ?st1*) $=_C$ *?s2*
  using *a1 a2* **by** *simp*
then have *config-to-istate* (*istate-to-config* (*istep tm ?st1*)) = *config-to-istate ?s2*
  using *config-to-istate-det*′ **by** *simp*
then have *istep tm* (*config-to-istate ?s1*) = *config-to-istate ?s2*
  using *st1-def istate-to-config-ident*′ **by** *simp*
then show *?thesis* **.**
qed

lemma *istep-correct*′:
  assumes *c1* ⊨⟨*tm*⟩= *c2*
  shows *istep tm* (*config-to-istate c1*) = *config-to-istate c2*
  using *assms istep-correct prod-cases3* **by** *metis*

lemma *config-eq-step0*:
  assumes *c1* $=_C$ *c2*
  shows *step0 c1 tm* $=_C$ *step0 c2 tm*
proof −
  have *config-to-istate c1* = *config-to-istate c2*
    using *config-to-istate-det*′ *assms* **by** *simp*
  then obtain *s* **where** *s-def1*: *s* = *config-to-istate c1*
      and *s-def2*: *s* = *config-to-istate c2*
    by *simp*

  obtain *c1*′ **where** *c1*′*-def*: *c1* ⊨⟨*tm*⟩= *c1*′
    by (*simp add*: *tm-step0-rel-def*)
  then have *step1*: *istep tm s* = *config-to-istate c1*′
    using *s-def1 istep-correct*′ **by** *simp*

  obtain *c2*′ **where** *c2*′*-def*: *c2* ⊨⟨*tm*⟩= *c2*′
    by (*simp add*: *tm-step0-rel-def*)
  then have *step2*: *istep tm s* = *config-to-istate c2*′
    using *s-def2 istep-correct*′ **by** *simp*

  have *config-to-istate c1*′ = *config-to-istate c2*′
    using *step1 step2* **by** *simp*
  then have *c1*′ $=_C$ *c2*′
    by (*metis config-to-istate-ident*′ *config-eq-trans*′ *config-eq-sym*′)
  with *c1*′*-def c2*′*-def* **show** *?thesis*
    by (*simp add*: *tm-step0-rel-def*)
qed

lemma *config-eq-steps0*: $c1 =_C c2 \Longrightarrow steps0\ c1\ tm\ n =_C steps0\ c2\ tm\ n$
by (*induction n*) (*use config-eq-step0* **in** *simp*)+

3 Constructing the IMP-Program

We now have a sufficient foundation to start constructing the IMP-program, that will later simulate an arbitrary Turing-Machine.

3.1 Translation: Intermediate State $\Longleftrightarrow$ IMP-State

First, we need to establish a mapping between our previously defined intermediate-state and an IMP-state. An IMP-state is mapping *vname* → *int* of variables to their values. Our IMP-program will need no more than five variables:

abbreviation *vnS* ≡ ''*tm-state*''

— Stores the current head symbol.
abbreviation *vnH* ≡ ''*tm-head*''

— Stores the infinite tape to left.
abbreviation *vnL* ≡ ''*tm-left*''

— Stores the infinite tape to right.
abbreviation *vnR* ≡ ''*tm-right*''

— This is a utility variable, which doesn't store any additional information, but will later be used as an index to execute the correct step.
abbreviation *vnSI* ≡ ''*tm-state-index*''

type-synonym *impstate* = *AExp.state*

Having this distinction of variables makes some proofs later a bit easier, but introduces the problem of invalid states. For example, the tapes are encoded as natural numbers, but the variables can have any integer, including negative numbers.

To preserve a unique and valid state, we introduce an invariant, which ensures the variables we use have a proper value.

abbreviation *impstate-inv* :: *impstate* ⇒ *bool* **where**
impstate-inv s ≡ ($s\ vnS \geq 0 \wedge (s\ vnH = 0 \vee s\ vnH = 1) \wedge s\ vnL \geq 0 \wedge s\ vnR \geq 0$)

Finally, we can define the translation between IMP-states and our intermediate-states:

abbreviation *istate-to-impstate* :: *impstate* ⇒ *istate* ⇒ *impstate* **where**
istate-to-impstate b st ≡ (
let (*s*, *l*, *h*, *r*) = *st in*
b (*vnS* := *int s*, *vnH* := *int* (*cell-to-nat h*), *vnL* := *int l*, *vnR* := *int r*)
)

abbreviation *impstate-to-istate* :: *impstate* ⇒ *istate* **where**
impstate-to-istate s ≡ (*nat* (*s vnS*), *nat* (*s vnL*), *nat-to-cell* (*nat* (*s vnH*)), *nat* (*s vnR*))

It can be shown, that every possible intermediate-state will always map to an IMP-state, that satisfies our previously defined invariant:

lemma *istate-to-impstate-inv*: *impstate-inv* (*istate-to-impstate b st*)
proof −
  obtain *s l h r* **where** *st* = (*s*, *l*, *h*, *r*)
    using *prod-cases4* **by** *blast*
  then show *?thesis*
    by (*cases h*) *simp+*
qed

Furthermore, we can also show that our translation is bijective:

lemma *istate-to-impstate-ident*: *impstate-to-istate* (*istate-to-impstate b st*) = *st*
proof −
  obtain *s l h r* **where** *st* = (*s*, *l*, *h*, *r*)
    using *prod-cases4* **by** *blast*
  then show *?thesis*
    by (*cases h*) *simp+*
qed

Using the previously established mapping between TM-configurations and intermediate-states, and the now defined mapping between intermediate-states and IMP-states, we finally define the chained mapping between TM-configurations and IMP-states:

abbreviation *config-to-impstate* :: *impstate* ⇒ *config* ⇒ *impstate* **where**
  config-to-impstate s c ≡ *istate-to-impstate s* (*config-to-istate c*)

abbreviation *impstate-to-config* :: *impstate* ⇒ *config* **where**
  impstate-to-config s ≡ *istate-to-config* (*impstate-to-istate s*)

3.2 Utility Programs

Now we can start constructing some smaller utility IMP-programs, which will slowly allow us to build the final IMP-program.

The arithmetic instructions provided by IMP are limited (only supporting addition), we will however construct programs to compute both multiplication by two and integer-division by two and its remainder, and prove them correct.

3.2.1 A Generalization for Hoare-Logic

First, we establish some facts about our Hoare logic, which will help us with some proofs later.

It is often much easier to prove that a program modifies a state in a given way t, such that if the starting state is s_0, the final state will be $t(s_0)$. However, making use of this is rather cumbersome. It is much easier to have the same statement, but with two predicates f and g, such that if f holds for every initial state s, then g also holds for $t(s)$, then f is valid pre-condition and g is a valid post-condition for the program. This allows us to insert arbitrary predicates and use the previously established fact of a fixed starting state s_0 much more easily.

lemma *hoare-generalize*:
  assumes $\bigwedge s_0.\ \vdash_t \{\lambda s.\ s = s_0\}\ c\ \{\lambda s.\ s = t\ s_0\}$

and $\bigwedge s.\ f\ s \Longrightarrow g\ (t\ s)$
shows $\vdash_t \{f\}\ c\ \{g\}$
proof $-$
have $\bigwedge s_0.\ \forall\, s.\ s = s_0 \longrightarrow (\exists\, s'.\ (c,s) \Rightarrow s' \wedge t\ s_0 = s')$
using *hoare-tvalid-def hoaret-sound-complete assms*(*1*) **by** *simp*
then have $\bigwedge s.\ (c,s) \Rightarrow t\ s$
by *simp*
then have $\forall\, s.\ f\ s \longrightarrow (\exists\, t.\ (c,s) \Rightarrow t \wedge g\ t)$
using *assms*(*2*) **by** *blast*
then show *?thesis*
using *hoare-tvalid-def hoaret-sound-complete* **by** *simp*
qed

While this formulation works well for most cases, it has a limitation: we have no assertion about the initial states to begin with. But sometimes we have proved statements about an initial state s_0, while posing some assumptions about it. To allow for this, we have to modify the above lemma a bit, by also asserting that the pre-condition f will only hold, if the assumptions posed on s_0 are also met.

lemma *hoare-generalize′*:
assumes $\bigwedge s_0.\ a\ s_0 \Longrightarrow \vdash_t \{\lambda s.\ s = s_0\}\ c\ \{\lambda s.\ s = t\ s_0\}$
and $\bigwedge s.\ f\ s \Longrightarrow g\ (t\ s)$ **and** $\bigwedge s.\ f\ s \Longrightarrow a\ s$
shows $\vdash_t \{f\}\ c\ \{g\}$
proof $-$
have $\bigwedge s_0.\ a\ s_0 \Longrightarrow \forall\, s.\ s = s_0 \longrightarrow (\exists\, s'.\ (c,s) \Rightarrow s' \wedge t\ s_0 = s')$
using *hoare-tvalid-def hoaret-sound-complete assms*(*1*) **by** *simp*
then have $\bigwedge s.\ a\ s \Longrightarrow (c,s) \Rightarrow t\ s$
by *simp*
then have $\forall\, s.\ f\ s \longrightarrow (\exists\, t.\ (c,s) \Rightarrow t \wedge g\ t)$
using *assms*(*2*) *assms*(*3*) **by** *blast*
then show *?thesis*
using *hoare-tvalid-def hoaret-sound-complete* **by** *simp*
qed

We now define two basic facts, which directly follow from the rules of the Hoare logic. However, we swapped the order of assumption, which makes proof automation easier later on.

lemma *Seq′*: $[\![\vdash_t \{P_2\}\ c_2\ \{P_3\};\ \vdash_t \{P_1\}\ c_1\ \{P_2\}\ ]\!] \Longrightarrow \vdash_t \{P_1\}\ c_1;;c_2\ \{P_3\}$
by (*simp only*: *Seq*)

lemma *conseq′*: $\vdash_t \{P\}\ c\ \{Q\} \Longrightarrow \forall\, s.\ P'\ s \longrightarrow P\ s \Longrightarrow \forall\, s.\ Q\ s \longrightarrow Q'\ s \Longrightarrow \vdash_t \{P'\}\ c\ \{Q'\}$
by (*simp only*: *conseq*)

lemma *partial-conseq′*: $\vdash \{P\}\ c\ \{Q\} \Longrightarrow \forall\, s.\ P'\ s \longrightarrow P\ s \Longrightarrow \forall\, s.\ Q\ s \longrightarrow Q'\ s \Longrightarrow \vdash \{P'\}\ c\ \{Q'\}$
by (*simp only*: *hoare.conseq*)

Another scenario we will encounter occasionally is where we have a pre-condition, which always will be False. We define two lemmas, which will make automating such scenarios rather easily, by only having to show that P is always False.

lemma *hoare-FalseI*: $\vdash_t \{\lambda s.\ False\}\ c\ \{Q\}$
by (*simp add*: *hoaret-sound-complete hoare-tvalid-def*)

lemma *hoare-Contr*: $\forall\, s.\ P'\ s \longrightarrow False \Longrightarrow \vdash_t \{P'\}\ c\ \{Q\}$
  by (*rule strengthen-pre*; *use hoare-FalseI* **in** *blast*)

3.2.2 Multiplication by 2

First, we construct a program `mul2`, which performs a multiplication by two:

$$mul2\ a\ b \quad \rightarrow \quad b\ :=\ 2 * a$$

This can be computed in a single step, by simply taking the sum of a and a again.

definition *mul2 a b* $\equiv$ (*b* ::= (*Plus* (*V a*) (*V a*)))

A proof of total-correctness is pretty straight-forward.

lemma *mul2-correct*:
  assumes $\bigwedge s.\ f\ s \Longrightarrow g\ (s\ (b := 2 * (s\ a)))$
  shows $\vdash_t \{f\}\ (mul2\ a\ b)\ \{g\}$
  unfolding *mul2-def* **by** (*rule Assign'*, *use assms* **in** *simp*)

lemma *mul2-correct'*: $\vdash_t \{\lambda s.\ s = s_0\}\ (mul2\ a\ b)\ \{\lambda s.\ s = s_0\ (b := 2 * (s_0\ a))\}$
  unfolding *mul2-def* **by** (*rule Assign'*, *simp*)

3.2.3 Integer-Division by 2

Next, we construct a program `moddiv2`, which performs integer-division by two, retrieving both the quotient and the remainder:

$$moddiv2\ a\ q\ m \quad \rightarrow \quad q\ :=\ a\ div\ 2,\ m\ :=\ a\ mod\ 2$$

Constructing such a program is bit more tedious and involves continuously subtracting in a WHILE-loop. The primitive version we implement also only works for positive numbers, however since our IMP-state invariant ensures that relevant variables are always positive, this is sufficient for our case.

definition *moddiv2-setup a q m* $\equiv$ (
  m ::= (*V a*) ;;
  q ::= (*N 0*)
)

lemma *moddiv2-setup-correct'*:
  assumes $q \neq m$
  shows $\vdash_t \{\lambda s.\ s = s_0\}\ moddiv2\text{-}setup\ a\ q\ m\ \{\lambda s.\ s = s_0\ (q := 0,\ m := s_0\ a)\}$
  unfolding *moddiv2-setup-def*
  by (*rule Seq'*; *rule Assign'*) (*use assms* **in** *force*)+

lemma *moddiv2-setup-correct*:
  assumes $q \neq m$ **and** $\bigwedge s.\ f\ s \Longrightarrow g\ (s\ (q := 0,\ m := s\ a))$
  shows $\vdash_t \{f\}\ moddiv2\text{-}setup\ a\ q\ m\ \{g\}$
proof –
  let $?t = \lambda s.\ s\ (q := 0,\ m := s\ a)$
  from *assms moddiv2-setup-correct'* **show** *?thesis*

using *hoare-generalize*[**where** t=$?t$] **by** *blast*
qed

definition *moddiv2-step q m* $\equiv$ (
  m ::= *Plus* (V m) (N (-2)) ;;
  q ::= *Plus* (V q) (N 1)
)

lemma *moddiv2-step-correct$'$*:
assumes $q \neq m$
shows $\vdash_t \{\lambda s.\ s = s_0\}$ *moddiv2-step q m* $\{\lambda s.\ s = s_0\ (q := s_0\ q + 1,\ m := s_0\ m - 2)\}$
unfolding *moddiv2-step-def*
by (*rule Seq$'$*; *rule Assign$'$*) (*use assms* **in** *force*)+

lemma *moddiv2-step-correct*:
assumes $q \neq m$ **and** $\bigwedge s.\ f\ s \Longrightarrow g\ (s\ (q := s\ q + 1,\ m := s\ m - 2))$
shows $\vdash_t \{f\}$ *moddiv2-step q m* $\{g\}$
proof $-$
let $?t = \lambda s.\ s\ (q := s\ q + 1,\ m := s\ m - 2)$
from *assms moddiv2-step-correct$'$* **show** *?thesis*
using *hoare-generalize*[**where** t=$?t$] **by** *blast*
qed

definition *moddiv2-loop q m* $\equiv$ (
WHILE Less (N 1) (V m) *DO* (
moddiv2-step q m
)
)

definition *moddiv2 a q m* $\equiv$ (
moddiv2-setup a q m ;;
moddiv2-loop q m
)

lemma *moddiv2-correct$'$*:
assumes $q \neq m$ **and** $s_0\ a \geq 0$
shows $\vdash_t \{\lambda s.\ s = s_0\}$ *moddiv2 a q m* $\{\lambda s.\ s = s_0\ (q := s_0\ a\ div\ 2,\ m := s_0\ a\ mod\ 2)\}$
proof $-$
let $?P = \lambda s.\ s = s_0$
let $?P' = \lambda s.\ \exists q'\ m'.\ s = s_0\ (q := q',\ m := m') \wedge s_0\ a = 2{*}q' + m' \wedge q' \geq 0 \wedge m' \geq 0$
let $?Q = \lambda s.\ s = s_0\ (q := s_0\ a\ div\ 2,\ m := s_0\ a\ mod\ 2)$

let $?f = \lambda n.\ \lambda s.\ ?P'\ s \wedge bval\ (Less\ (N\ 1)\ (V\ m))\ s \wedge n = nat\ (s\ m)$
let $?g = \lambda n.\ \lambda s.\ ?P'\ s \wedge nat\ (s\ m) < n$
have *step*: $\bigwedge n{::}nat.\ \vdash_t \{?f\ n\}$ *moddiv2-step q m* $\{?g\ n\}$
proof $-$
fix n :: *nat*
have $\bigwedge s.\ ?f\ n\ s \Longrightarrow ?g\ n\ (s\ (q := s\ q + 1,\ m := s\ m - 2))$
using *assms* **by** *fastforce*

then show $\vdash_t$ $\{?f\ n\}$ *moddiv2-step q m* $\{?g\ n\}$
using *assms(1) moddiv2-step-correct* **by** *presburger*
qed

have *loop-body*: $\vdash_t$ $\{?P'\}$ *moddiv2-loop q m* $\{?Q\}$
unfolding *moddiv2-loop-def*
by (*rule While-fun'*[**where** $f=\lambda s.\ nat\ (s\ m)$], *use step* **in** *auto*)

have *loop-setup*: $\vdash_t$ $\{?P\}$ *moddiv2-setup a q m* $\{?P'\}$
unfolding *moddiv2-setup-def*
by (*rule Seq'*; *rule Assign'*) (*use assms* **in** *force*)+

show *?thesis*
unfolding *moddiv2-def*
by (*rule Seq*; *use loop-setup loop-body* **in** *simp*)
qed

lemma *moddiv2-correct*:
assumes $q \neq m$
and $\bigwedge s.\ f\ s \Longrightarrow g\ (s\ (q := s\ a\ div\ 2,\ m := s\ a\ mod\ 2))$
and $\bigwedge s.\ f\ s \Longrightarrow s\ a \geq 0$
shows $\vdash_t$ $\{f\}$ (*moddiv2 a q m*) $\{g\}$
proof $-$
let $?a = \lambda s.\ s\ a \geq 0$
let $?t = \lambda s.\ s\ (q := s\ a\ div\ 2,\ m := s\ a\ mod\ 2)$
from *assms moddiv2-correct'* **show** *?thesis*
using *hoare-generalize'*[**where** $a=?a$ **and** $t=?t$] **by** *blast*
qed

3.2.4 List-Index Program

abbreviation *eq a b* $\equiv$ *And* (*Not* (*Less a b*)) (*Not* (*Less b a*))
abbreviation *neq a b* $\equiv$ *Not* (*eq a b*)

Next, we need to construct a program that simulates a (non-fallthrough) `SWITCH`-statement. Precisely, we construct a program `list_index_prog`, that takes in a variable name, a list of programs and a fallback program, and constructs a new program that uses the provided variable to index the program-list and execute the right one, or execute the fallback program if the index is out of bounds.

Constructing the program in a way, that doesn't change the variables (e.g. by subtracting one from the index at every step), requires us to introduce an index-offset:

fun *list-index-prog'* :: *vname* $\Rightarrow$ *int* $\Rightarrow$ *com list* $\Rightarrow$ *com* $\Rightarrow$ *com* **where**
list-index-prog' vn n (*p#ps*) *e* = (
IF eq (*V vn*) (*N n*) *THEN p*
ELSE list-index-prog' vn (*n+1*) *ps e*
) |
list-index-prog' vn n [] *e* = *e*

We can now prove that our program with index-offset works as expected, by executing the program at the index or the fallback program. Proving this requires delicate use of induction, but then yields a proper proof:

lemma *list-index-prog′-skip*:
  assumes $i \geq$ *length ps*
    and $\vdash_t \{P\}\ e\ \{Q\}$
  shows $\vdash_t \{\lambda s.\ P\ s \wedge s\ vn = n + int\ i\}$ *list-index-prog′ vn n ps e* $\{Q\}$
using *assms* **proof** (*induction vn n ps e arbitrary*: *i rule*: *list-index-prog′.induct*)
  — We still are checking indices:
  — Show that they mismatch and continue with induction.
  case (*1 vn n p ps e*)

  let $?TP' = \lambda s.\ P\ s \wedge s\ vn = n + int\ i \wedge bval\ (eq\ (V\ vn)\ (N\ n))\ s$
  have $\forall s.\ ?TP'\ s \longrightarrow False$
    using *1*(*2*) **by** *simp*
  moreover have $\vdash_t \{\lambda s.\ False\}\ p\ \{Q\}$
    using *hoare-FalseI* **by** *simp*
  ultimately have *TrueCase*: $\vdash_t \{?TP'\}\ p\ \{Q\}$
    using *strengthen-pre*[**where** $P{=}\lambda s.\ False$ **and** $P'{=}?TP'$] **by** *simp*

  let $?FP = \lambda s.\ P\ s \wedge s\ vn = n + 1 + int\ (i - 1)$
  let $?FP' = \lambda s.\ P\ s \wedge s\ vn = n + int\ i \wedge \neg\ bval\ (eq\ (V\ vn)\ (N\ n))\ s$
  have *length ps* $\leq i - 1$
    using *1*(*2*) **by** *simp*
  then have $\vdash_t \{?FP\}$ *list-index-prog′ vn* $(n + 1)$ *ps e* $\{Q\}$
    using *1*(*1*)[*of* $i{-}1$] *1*(*3*) **by** *simp*
  moreover have $\forall s.\ ?FP'\ s \longrightarrow ?FP\ s$
    by *fastforce*
  ultimately have *FalseCase*: $\vdash_t \{?FP'\}$ *list-index-prog′ vn* $(n + 1)$ *ps e* $\{Q\}$
    using *strengthen-pre*[**where** $P{=}?FP$ **and** $P'{=}?FP'$] **by** *simp*

  from *list-index-prog′.simps*(*1*) **show** *?case*
    using *TrueCase FalseCase* **by** (*simp add*: *If*)
next
  — Index is out of bounds, finally execute the fallback.
  case (*2 vn n e*)

  let $?P' = \lambda s.\ P\ s \wedge s\ vn = n + int\ i$
  have $\forall s.\ ?P'\ s \longrightarrow P\ s$
    by *simp*
  with *2* **show** *?case*
    using *conseq*[**where** $P'{=}?P'$ **and** $P{=}P$] **by** *simp*
qed

lemma *list-index-prog′-correct*:
  assumes $i <$ *length ps*
    and $\vdash_t \{P\}\ ps!i\ \{Q\}$
  shows $\vdash_t \{\lambda s.\ P\ s \wedge s\ vn = n + int\ i\}$ *list-index-prog′ vn n ps e* $\{Q\}$
using *assms* **proof** (*induction vn n ps e arbitrary*: *i rule*: *list-index-prog′.induct*)
  case (*1 vn n p ps e*)
  let $?P = \lambda s.\ P\ s \wedge s\ vn = n + int\ i$

```
show ?case proof (cases i = 0)
  — Index match! Execute our branch.
  case True

  let ?TP = λs. ?P s ∧ bval (eq (V vn) (N n)) s
  have ∀ s. ?TP s ⟶ P s
    by simp
  moreover have ⊢t {P} p {Q}
    using 1(3) True by simp
  ultimately have TrueCase: ⊢t {?TP} p {Q}
    using strengthen-pre[where P=P and P′=?TP] by simp

  let ?FP = λs. ?P s ∧ ¬ bval (eq (V vn) (N n)) s
  have ∀ s. ?FP s ⟶ False
    using True by simp
  moreover have ⊢t {λs. False} list-index-prog′ vn (n + 1) ps e {Q}
    using hoare-FalseI by simp
  ultimately have FalseCase: ⊢t {?FP} list-index-prog′ vn (n + 1) ps e {Q}
    using strengthen-pre[where P=λs. False and P′=?FP] by simp

  from list-index-prog′.simps(1) show ?thesis
    using TrueCase FalseCase by (simp add: If)
next
  — Index mismatch, continue with induction.
  case False

  let ?TP = λs. ?P s ∧ bval (eq (V vn) (N n)) s
  have ∀ s. ?TP s ⟶ False
    using False by simp
  moreover have ⊢t {λs. False} p {Q}
    using hoare-FalseI by simp
  ultimately have TrueCase: ⊢t {?TP} p {Q}
    using strengthen-pre[where P=λs. False and P′=?TP] by simp

  let ?FP = λs. ?P s ∧ ¬ bval (eq (V vn) (N n)) s
  let ?FP′ = λs. P s ∧ s vn = n + 1 + int (i − 1)
  have i − 1 < length ps
    using 1(2) False by simp
  moreover have ⊢t {P} ps ! (i − 1) {Q}
    using 1(3) False by simp
  ultimately have ⊢t {?FP′} list-index-prog′ vn (n + 1) ps e {Q}
    using 1(1) by simp
  moreover have ∀ s. ?FP s ⟶ ?FP′ s
    by fastforce
  ultimately have FalseCase: ⊢t {?FP} list-index-prog′ vn (n + 1) ps e {Q}
    using strengthen-pre[where P=?FP′ and P′=?FP] by simp

  from list-index-prog′.simps(1) show ?thesis
    using TrueCase FalseCase by (simp add: If)
```

qed
next
case (*2 vn n*)
then show *?case* **by** *simp*
qed

The final `list_index_prog` program is now the above defined program, with an index-offset of $n = 0$.

definition *list-index-prog* :: *vname* ⇒ *com list* ⇒ *com* ⇒ *com* **where**
list-index-prog vn ps e = *list-index-prog′ vn 0 ps e*

The same proofs for the final program now follow directly:

corollary *list-index-prog-skip*:
assumes $i \geq$ *length ps* **and** $\vdash_t$ $\{P\}$ *e* $\{Q\}$
shows $\vdash_t$ $\{\lambda s.\ P\ s \wedge s\ vn = int\ i\}$ *list-index-prog vn ps e* $\{Q\}$
unfolding *list-index-prog-def*
using *assms list-index-prog′-skip*[**where** *n=0*] **by** *simp*

corollary *list-index-prog-correct*:
assumes $i <$ *length ps* **and** $\vdash_t$ $\{P\}$ *ps*!*i* $\{Q\}$
shows $\vdash_t$ $\{\lambda s.\ P\ s \wedge s\ vn = int\ i\}$ *list-index-prog vn ps e* $\{Q\}$
unfolding *list-index-prog-def*
using *assms list-index-prog′-correct*[**where** *n=0*] **by** *simp*

3.3 Operations

We now start to construct programs that compute the possible Turing-Machine operations:

- WB: Write Bk to head
- WO: Write Oc to head
- L: Push head to right tape, pop left tape to head
- R: Push head to left tape, pop right tape to head
- Nop: Do nothing

From now, we will encounter the following abbreviations in practically every proof. They both make statements about an IMP-state and ensure its invariant still holds. The first also states, that the IMP-state contains the same information as an intermediate-state, while the latter states, the IMP-state contains the same information as a TM-configuration.

abbreviation *impstate-to-istate-inv* :: *impstate* ⇒ *istate* ⇒ *bool* (**infix** $\rightarrow_S$ *55*) **where**
$s \rightarrow_S st \equiv$ *impstate-inv s* ∧ *impstate-to-istate s* = *st*

abbreviation *impstate-to-config-inv* :: *impstate* ⇒ *config* ⇒ *bool* (**infix** $\rightarrow_C$ *55*) **where**
$s \rightarrow_C c \equiv$ *impstate-inv s* ∧ (*impstate-to-config s* $=_C$ *c*)

We now show some useful rules about the abbreviations.

An important fact is that, although we have defined our abbreviation to be a relation, it actually still works almost like a function, in terms that its output is determined. A state can at most represent one configuration. However due to ambigiuous definition as explained in section 2.1.1, it doesn't imply real equivalence, but only our custom $=_C$ equivalence relation.

lemma *impstate-to-config-inv-det*:
  assumes $s \rightarrow_C (s1, l1, r1)$ **and** $s \rightarrow_C (s2, l2, r2)$
  shows $(s1, l1, r1) =_C (s2, l2, r2)$
  using *assms config-eq-sym′ config-eq-trans′* **by** *blast*

lemma *impstate-to-config-inv-det′*:
  assumes $s \rightarrow_C c1$ **and** $s \rightarrow_C c2$
  shows $c1 =_C c2$
proof −
  — For some reason a straight-forward metis proof would take too much time here.
  obtain *s1 l1 r1* **where** *c1-def*: $c1 = (s1, l1, r1)$
    using *prod-cases3* **by** *blast*
  obtain *s2 l2 r2* **where** *c2-def*: $c2 = (s2, l2, r2)$
    using *prod-cases3* **by** *blast*
  from *c1-def c2-def* **show** *?thesis*
    using *assms impstate-to-config-inv-det* **by** *blast*
qed

lemma *config-eq-implies-istate-eq*:
  assumes $s \rightarrow_C c$
  shows $s \rightarrow_S$ *config-to-istate c*
  using *assms*
proof −
  have *config-to-istate* (*impstate-to-config s*) = *impstate-to-istate s*
    using *istate-to-config-ident′* **by** *blast*
  moreover have *impstate-to-config s* $=_C$ *c*
    using *assms* **by** *blast*
  ultimately have *impstate-to-istate s* = *config-to-istate c*
    using *config-to-istate-det′*[*of impstate-to-config s*] **by** *presburger*
  thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *config-eq-implies-istate-eq′*:
  assumes $s \rightarrow_C$ *istate-to-config st*
  shows $s \rightarrow_S st$
  using *assms*
proof −
  obtain *c* **where** $s \rightarrow_C c \wedge$ *config-to-istate c* = *st*
    using *assms istate-to-config-ident′* **by** *blast*
  thus *?thesis* **using** *config-eq-implies-istate-eq* **by** *blast*
qed

lemma *istate-eq-implies-config-eq*:
  assumes $s \rightarrow_S$ *config-to-istate c*
  shows $s \rightarrow_C c$
  by (*simp add*: *assms config-to-istate-ident′*)

lemma *istate-eq-implies-config-eq'*:
assumes $s \rightarrow_S st$
shows $s \rightarrow_C$ *istate-to-config st*
by (*simp add*: *assms config-eq-refl'*)

With this foundation, we now construct a program for each of the possible TM-operations.

3.3.1 Write-Bk

definition *prog-WB* :: *state* ⇒ *com* **where**
prog-WB s ≡ (
vnS ::= *N* (*int s*) ;;
vnH ::= *N* (*int* (*cell-to-nat Bk*))
)

lemma *prog-WB-hoare*:
$\vdash_t$ $\{\lambda s.\ s \rightarrow_S (st, l, h, r)\}$ *prog-WB st'* $\{\lambda s.\ s \rightarrow_S$ *iupdate* $(WB, st')\ (st, l, Bk, r)\}$
unfolding *prog-WB-def* **by** (*rule Seq*[**where** $P_2{=}\lambda s.\ s \rightarrow_S (st', l, h, r)$]; *rule Assign'*, *simp*)

3.3.2 Write-Oc

definition *prog-WO* :: *state* ⇒ *com* **where**
prog-WO s ≡ (
vnS ::= *N* (*int s*) ;;
vnH ::= *N* (*int* (*cell-to-nat Oc*))
)

lemma *prog-WO-hoare*:
$\vdash_t$ $\{\lambda s.\ s \rightarrow_S (st, l, h, r)\}$ *prog-WO st'* $\{\lambda s.\ s \rightarrow_S$ *iupdate* $(WO, st')\ (st, l, Oc, r)\}$
unfolding *prog-WO-def* **by** (*rule Seq*[**where** $P_2{=}\lambda s.\ s \rightarrow_S (st', l, h, r)$]; *rule Assign'*, *simp*)

3.3.3 Move-Left

definition *prog-L* :: *state* ⇒ *com* **where**
prog-L s ≡ (
vnS ::= *N* (*int s*) ;;
mul2 vnR vnR ;;
vnR ::= *Plus* (*V vnR*) (*V vnH*) ;;
moddiv2 vnL vnL vnH
)

lemma *prog-L-hoare*:
$\vdash_t$ $\{\lambda s.\ s \rightarrow_S (st, l, h, r)\}$ *prog-L st'* $\{\lambda s.\ s \rightarrow_S$ *iupdate* $(L, st')\ (st, l, h, r)\}$
unfolding *prog-L-def* **proof** (*rule Seq*[**where** $P_2{=}\lambda s.\ s \rightarrow_S (st', l, h, 2{*}r + \textit{cell-to-nat}\ h)$])
have $\vdash_t$
$\{\lambda s.\ s \rightarrow_S (st, l, h, r)\}$
vnS ::= *N* (*int st'*)
$\{\lambda s.\ s \rightarrow_S (st', l, h, r)\}$
by (*rule Assign'*, *simp*)
moreover have $\vdash_t$
$\{\lambda s.\ s \rightarrow_S (st', l, h, r)\}$

    mul2 vnR vnR
    $\{\lambda s.\ s \rightarrow_S (st', l, h, 2{*}r)\}$
  proof −
    let $?f = \lambda s.\ s \rightarrow_S (st', l, h, r)$
    let $?g = \lambda s.\ s \rightarrow_S (st', l, h, 2{*}r)$
    have $\bigwedge s.\ ?f\ s \Longrightarrow\ ?g\ (s\ (vnR := 2 * s\ vnR))$ **by** *force*
    then show *?thesis* **using** *mul2-correct* **by** *presburger*
  qed
  moreover have $\vdash_t$
    $\{\lambda s.\ s \rightarrow_S (st', l, h, 2{*}r)\}$
    $vnR ::= Plus\ (V\ vnR)\ (V\ vnH)$
    $\{\lambda s.\ s \rightarrow_S (st', l, h, 2{*}r + cell\text{-}to\text{-}nat\ h)\}$
    by (*rule Assign′*, *auto*)
  ultimately show $\vdash_t$
    $\{\lambda s.\ s \rightarrow_S (st, l, h, r)\}$
    $vnS ::= N\ (int\ st');;\ mul2\ vnR\ vnR\ ;;\ vnR ::= Plus\ (V\ vnR)\ (V\ vnH)$
    $\{\lambda s.\ s \rightarrow_S (st', l, h, 2{*}r + cell\text{-}to\text{-}nat\ h)\}$
    using *Seq* **by** *blast*
next
  have $\vdash_t$
    $\{\lambda s.\ s \rightarrow_S (st', l, h, 2{*}r + cell\text{-}to\text{-}nat\ h)\}$
    moddiv2 vnL vnL vnH
    $\{\lambda s.\ s \rightarrow_S (st', l\ div\ 2, nat\text{-}to\text{-}cell\ (l\ mod\ 2), 2{*}r + cell\text{-}to\text{-}nat\ h)\}$
  proof −
    let $?f = \lambda s.\ s \rightarrow_S (st', l, h, 2{*}r + cell\text{-}to\text{-}nat\ h)$
    let $?g = \lambda s.\ s \rightarrow_S (st', l\ div\ 2, nat\text{-}to\text{-}cell\ (l\ mod\ 2), 2{*}r + cell\text{-}to\text{-}nat\ h)$
    have $\bigwedge s.\ ?f\ s \Longrightarrow\ ?g\ (s\ (vnL := s\ vnL\ div\ 2, vnH := s\ vnL\ mod\ 2))$
    proof −
      fix $s :: impstate$
      assume $?f\ s$
      define s' **where** s'-*def*: $s' = s\ (vnL := s\ vnL\ div\ 2, vnH := s\ vnL\ mod\ 2)$

      from ‹$?f\ s$› **have** $s\ vnS = int\ st'$
        by *force*
      then have s'-*st*: $s'\ vnS = int\ st'$ **and** $s'\ vnS \geq 0$
        using s'-*def* **by** *simp+*

      from ‹$?f\ s$› **have** *l-val*: $s\ vnL = int\ l$
        by *force*
      then have s'-*l*: $s'\ vnL = int\ (l\ div\ 2)$ **and** $s'\ vnL \geq 0$
        using s'-*def* **by** *simp+*

      from ‹$?f\ s$› **have** *r-val*: $s\ vnR = int\ (2{*}r + cell\text{-}to\text{-}nat\ h)$
        by *force*
      then have s'-*r*: $s'\ vnR = int\ (2{*}r + cell\text{-}to\text{-}nat\ h)$ **and** $s'\ vnR \geq 0$
        using s'-*def* **by** *simp+*

      from ‹$?f\ s$› **have** $s\ vnH = int\ (cell\text{-}to\text{-}nat\ h)$
        by *force*

then have s'-*h*: $s'\ vnH = int\ (l\ mod\ 2)$
using s'-*def* *l-val* **by** (*simp add*: *zmod-int*)
then have s'-*h-invar*: $s'\ vnH = 0 \vee s'\ vnH = 1$
by *linarith*

have $?g\ s'$
using s'-*st* s'-*l* s'-*h* s'-*h-invar* s'-*r* **by** *simp*
then show $?g\ (s\ (vnL := s\ vnL\ div\ 2,\ vnH := s\ vnL\ mod\ 2))$
using s'-*def* **by** *blast*
qed
then show *?thesis*
using *moddiv2-correct* **by** *fastforce*
qed
then show $\vdash_t$
$\{\lambda s.\ s \rightarrow_S (st',\ l,\ h,\ 2{*}r + cell\text{-}to\text{-}nat\ h)\}$
moddiv2 vnL vnL vnH
$\{\lambda s.\ s \rightarrow_S iupdate\ (L,\ st')\ (st,\ l,\ h,\ r)\}$
by *simp*
qed

3.3.4 Move-Right

definition *prog-R* :: *state* $\Rightarrow$ *com* **where**
prog-R $s \equiv$ (
$vnS ::= N\ (int\ s)$;;
mul2 vnL vnL ;;
$vnL ::= Plus\ (V\ vnL)\ (V\ vnH)$;;
moddiv2 vnR vnR vnH
)

lemma *prog-R-hoare*:
$\vdash_t \{\lambda s.\ s \rightarrow_S (st,\ l,\ h,\ r)\}\ prog\text{-}R\ st'\ \{\lambda s.\ s \rightarrow_S iupdate\ (R,\ st')\ (st,\ l,\ h,\ r)\}$
unfolding *prog-R-def* **proof** (*rule Seq*[**where** $P_2 = \lambda s.\ s \rightarrow_S (st',\ 2{*}l + cell\text{-}to\text{-}nat\ h,\ h,\ r)$])
have $\vdash_t$
$\{\lambda s.\ s \rightarrow_S (st,\ l,\ h,\ r)\}$
$vnS ::= N\ (int\ st')$
$\{\lambda s.\ s \rightarrow_S (st',\ l,\ h,\ r)\}$
by (*rule Assign'*, *simp*)
moreover have $\vdash_t$
$\{\lambda s.\ s \rightarrow_S (st',\ l,\ h,\ r)\}$
mul2 vnL vnL
$\{\lambda s.\ s \rightarrow_S (st',\ 2{*}l,\ h{,}r)\}$
proof $-$
let $?f = \lambda s.\ s \rightarrow_S (st',\ l,\ h,\ r)$
let $?g = \lambda s.\ s \rightarrow_S (st',\ 2{*}l,\ h,\ r)$
have $\bigwedge s.\ ?f\ s \Longrightarrow ?g\ (s\ (vnL := 2 * s\ vnL))$ **by** *force*
then show *?thesis* **using** *mul2-correct* **by** *presburger*
qed
moreover have $\vdash_t$
$\{\lambda s.\ s \rightarrow_S (st',\ 2{*}l,\ h,\ r)\}$

```
    vnL ::= Plus (V vnL) (V vnH)
    {λs. s →_S (st', 2*l + cell-to-nat h, h, r)}
    by (rule Assign', auto)
  ultimately show ⊢_t
    {λs. s →_S (st, l, h, r)}
    vnS ::= N (int st');; mul2 vnL vnL ;; vnL ::= Plus (V vnL) (V vnH)
    {λs. s →_S (st', 2*l + cell-to-nat h, h, r)}
    using Seq by blast
next
  have ⊢_t
    {λs. s →_S (st', 2*l + cell-to-nat h, h, r)}
    moddiv2 vnR vnR vnH
    {λs. s →_S (st', 2*l + cell-to-nat h, nat-to-cell (r mod 2), r div 2)}
  proof −
    let ?f = λs. s →_S (st', 2*l + cell-to-nat h, h, r)
    let ?g = λs. s →_S (st', 2*l + cell-to-nat h, nat-to-cell (r mod 2), r div 2)
    have ⋀s. ?f s ⟹ ?g (s (vnR := s vnR div 2, vnH := s vnR mod 2))
    proof −
      fix s :: impstate
      assume ?f s
      define s' where s'-def: s' = s (vnR := s vnR div 2, vnH := s vnR mod 2)

      from ‹?f s› have s vnS = int st'
        by force
      then have s'-st: s' vnS = int st' and s' vnS ≥ 0
        using s'-def by simp+

      from ‹?f s› have l-val: s vnL = int (2*l + cell-to-nat h)
        by force
      then have s'-l: s' vnL = int (2*l + cell-to-nat h) and s' vnL ≥ 0
        using s'-def by simp+

      from ‹?f s› have r-val: s vnR = int r
        by force
      then have s'-r: s' vnR = int (r div 2) and s' vnR ≥ 0
        using s'-def by simp+

      from ‹?f s› have s vnH = int (cell-to-nat h)
        by force
      then have s'-h: s' vnH = int (r mod 2)
        using s'-def r-val by (simp add: zmod-int)
      then have s'-h-invar: s' vnH = 0 ∨ s' vnH = 1
        by linarith

      have ?g s'
        using s'-st s'-l s'-h s'-h-invar s'-r by simp
      then show ?g (s (vnR := s vnR div 2, vnH := s vnR mod 2))
        using s'-def by blast
    qed
```

then show *?thesis*
using *moddiv2-correct* **by** *fastforce*
qed
then show $\vdash_t$
$\{\lambda s.\ s \rightarrow_S (st',\ 2{*}l + cell\text{-}to\text{-}nat\ h,\ h,\ r)\}$
moddiv2 vnR vnR vnH
$\{\lambda s.\ s \rightarrow_S iupdate\ (R,\ st')\ (st,\ l,\ h,\ r)\}$
by *simp*
qed

3.3.5 No Operation

definition *prog-Nop* :: *state* ⇒ *com* **where**
prog-Nop s ≡ *vnS* ::= *N* (*int s*)

lemma *prog-Nop-hoare*:
$\vdash_t \{\lambda s.\ s \rightarrow_S (st,\ l,\ h,\ r)\}$ *prog-Nop st'* $\{\lambda s.\ s \rightarrow_S iupdate\ (Nop,\ st')\ (st,\ l,\ h,\ r)\}$
unfolding *prog-Nop-def* **by** (*rule Assign'*, *simp*)

3.3.6 Pattern-Matching Operation

Now we can construct a pattern-matching wrapper, which provides the correct program, given a specific instruction.

fun *prog-step* :: *instr* ⇒ *com* **where**
prog-step (*WB*, *st*) = *prog-WB st* |
prog-step (*WO*, *st*) = *prog-WO st* |
prog-step (*L*, *st*) = *prog-L st* |
prog-step (*R*, *st*) = *prog-R st* |
prog-step (*Nop*, *st*) = *prog-Nop st*

The proofs again follow directly.

corollary *prog-step-hoare*:
$\vdash_t \{\lambda s.\ s \rightarrow_S (st,\ l,\ h,\ r)\}$ *prog-step* (*a*, *st'*) $\{\lambda s.\ s \rightarrow_S iupdate\ (a,\ st')\ (st,\ l,\ h,\ r)\}$
proof (*cases a*)
case *WB* **thus** *?thesis* **using** *prog-WB-hoare* **by** *simp*
next
case *WO* **thus** *?thesis* **using** *prog-WO-hoare* **by** *simp*
next
case *L* **thus** *?thesis* **using** *prog-L-hoare* **by** *simp*
next
case *R* **thus** *?thesis* **using** *prog-R-hoare* **by** *simp*
next
case *Nop* **thus** *?thesis* **using** *prog-Nop-hoare* **by** *simp*
qed

corollary *prog-step-hoare'*:
$\vdash_t \{\lambda s.\ s \rightarrow_S st\}$ *prog-step i* $\{\lambda s.\ s \rightarrow_S iupdate\ i\ st\}$
proof −
obtain *s l h r* **where** *st* = (*s*, *l*, *h*, *r*)
using *prod-cases4* **by** *blast*

moreover obtain *a st′* **where** $i = (a,\ st')$
  by *fastforce*
ultimately show *?thesis*
  using *prog-step-hoare* **by** *simp*
qed

3.3.7 State-Index Program

In order to execute the correct step for each state, which create a list containing all possible transitions. We then compute the correct index depending on the current state and head symbol, and using our previously constructed `list_index_prog` jump to the correct step program.

We now construct a program `prog_SI`, that computes this index:

$$prog\_SI \quad \to \quad si\ :=\ 2 * s + h$$

definition *prog-SI* :: *com* **where**
  $\textit{prog-SI} \equiv \textit{vnSI} ::= \textit{Plus}\ (\textit{Plus}\ (V\ \textit{vnS})\ (V\ \textit{vnS}))\ (V\ \textit{vnH})$

The proof is again rather straight-forward.

lemma *prog-SI-correct*:
  $\vdash_t \{\lambda s.\ s \to_S st\}\ \textit{prog-SI}\ \{\lambda s.\ s \to_S st \wedge s\ \textit{vnSI} = \textit{int}\ (\textit{istate-to-index}\ st)\}$
  unfolding *prog-SI-def* **by** (*rule Assign′*, *auto*)

3.4 Single-Step Program

We can now finally construct a program that, given any Turing-Machine, executes the next step depending on the current state. Although we have built a good foundation, the core proof will still rather large.

First, we construct a list of IMP-programs from a Turing-Machine. The list contains all the correct instructions, mapped to their corresponding index. Since $s = 0$ is the final state, all following steps are Nop instructions. Therefore, we append the list with two Nop instructions, so that we can still use the simple $i = 2 * s + h$ formula to get the index.

definition *tm-to-step-progs* :: *tprog0* ⇒ *com list* **where**
  tm-to-step-progs tm = (
    let n = (*Nop*, *0*) *in*
    map prog-step (*n*#*n*#*tm*)
)

Proving this table to be correct is easy.

lemma *tm-to-step-progs-hoare*:
  assumes $i < \textit{length}\ (\textit{tm-to-step-progs}\ tm)$
  shows $\vdash_t \{\lambda s.\ s \to_S st\}\ (\textit{tm-to-step-progs}\ tm)!i\ \{\lambda s.\ s \to_S \textit{iupdate}\ (tm\ @_I\ i)\ st\}$
proof −
  have $(\textit{tm-to-step-progs}\ tm)!i = \textit{prog-step}\ (tm\ @_I\ i)$
    using *assms tm-to-step-progs-def*
    by (*auto*, *use numeral-2-eq-2* **in** *argo*, *simp add*: *nth-Cons′*)
  with *prog-step-hoare′* **show** *?thesis* **by** *simp*
qed

For the final `tm_imp_step` program, we chain the computation of the instruction-index together with a `list_index_prog`, with a table of instructions and the computed index.

definition *tm-imp-step* :: *tprog0* ⇒ *com* **where**
  tm-imp-step p = (
    prog-SI ;;
    list-index-prog vnSI (*tm-to-step-progs p*) (*prog-step* (*Nop*, *0*))
)

While arguing that, given the current facts, our `tm_imp_step` behaves as expected might seem obvious, formally verifying this is not as trivial.

lemma *tm-imp-step-correct-aux*:
  $\vdash_t$ {λ*s*. *s* $\rightarrow_S$ *st*} *tm-imp-step tm* {λ*s*. *s* $\rightarrow_S$ *istep tm st*}
  unfolding *tm-imp-step-def*
proof (*rule Seq*[**where** P_2=λ*s*. (*s* $\rightarrow_S$ *st*) ∧ *s vnSI* = *int* (*istate-to-index st*)])
  show $\vdash_t$
    {λ*s*. *s* $\rightarrow_S$ *st*}
    prog-SI
    {λ*s*. (*s* $\rightarrow_S$ *st*) ∧ *s vnSI* = *int* (*istate-to-index st*)}
    using *prog-SI-correct* **by** *blast*
next
  let *?i* = *istate-to-index st*
  let *?ps* = *tm-to-step-progs tm*
  show $\vdash_t$
    {λ*s*. (*s* $\rightarrow_S$ *st*) ∧ *s vnSI* = *int* (*istate-to-index st*)}
    list-index-prog vnSI ?ps (*prog-step* (*Nop*, *0*))
    {λ*s*. *s* $\rightarrow_S$ *istep tm st*}
  proof (*cases ?i* < *length ?ps*)
    — Index is in-bounds, execute the corresponding instruction.
    case *True*
    then have $\vdash_t$
      {λ*s*. *s* $\rightarrow_S$ *st*}
      ?ps!*?i*
      {λ*s*. *s* $\rightarrow_S$ *iupdate* (*tm* $@_I$ *?i*) *st*}
      using *tm-to-step-progs-hoare* **by** *blast*
    with *True* **have** $\vdash_t$
      {λ*s*. *s* $\rightarrow_S$ *st* ∧ *s vnSI* = *int* (*istate-to-index st*)}
      list-index-prog vnSI ?ps (*prog-step* (*Nop*, *0*))
      {λ*s*. *s* $\rightarrow_S$ *iupdate* (*tm* $@_I$ *?i*) *st*}
      using *list-index-prog-correct*[**where** *ps*=*?ps* **and** *i*=*?i*
          and *P*=λ*s*. *s* $\rightarrow_S$ *st*
          and *Q*=λ*s*. *s* $\rightarrow_S$ *iupdate* (*tm* $@_I$ *?i*) *st*]
      by *simp*
    moreover have *iupdate* (*tm* $@_I$ *?i*) *st* = *istep tm st*
      using *True istep-index-correct′ tm-to-step-progs-def* **by** *simp*
    ultimately show *?thesis* **by** *simp*
  next
    — Index is out-of-bounds, execute NOP-instruction instead.
    case *False*
    have $\vdash_t$

$\{\lambda s.\ s \rightarrow_S st\}$
prog-step (*Nop*, *0*)
$\{\lambda s.\ s \rightarrow_S iupdate\ (Nop,\ 0)\ st\}$
using *prog-step-hoare′* **by** *blast*
with *False* **have** $\vdash_t$
$\{\lambda s.\ s \rightarrow_S st \wedge s\ vnSI = int\ (istate\text{-}to\text{-}index\ st)\}$
list-index-prog vnSI ?ps (*prog-step* (*Nop*, *0*))
$\{\lambda s.\ s \rightarrow_S iupdate\ (Nop,\ 0)\ st\}$
using *list-index-prog-skip*[**where** *ps*=*?ps* **and** *i*=*?i*
and $P=\lambda s.\ s \rightarrow_S st$
and $Q=\lambda s.\ s \rightarrow_S iupdate\ (Nop,\ 0)\ st$]
by *simp*
moreover have *iupdate* (*Nop*, *0*) *st* = *istep tm st*
using *False istep-index-skip′ tm-to-step-progs-def* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
qed

lemma *tm-imp-step-correct*:
assumes (*s1*, *l1*, *r1*) $\models\langle tm\rangle=$ (*s2*, *l2*, *r2*)
shows $\vdash_t$ $\{\lambda s.\ s \rightarrow_C (s1,\ l1,\ r1)\}$ *tm-imp-step tm* $\{\lambda s.\ s \rightarrow_C (s2,\ l2,\ r2)\}$
proof −
let *?st1* = *config-to-istate* (*s1*, *l1*, *r1*)
have *a1*: $\forall s.\ (s \rightarrow_C (s1,\ l1,\ r1)) \longrightarrow (s \rightarrow_S ?st1)$
using *config-eq-implies-istate-eq* **by** *blast*

let *?st2* = *config-to-istate* (*s2*, *l2*, *r2*)
have *istep tm ?st1* = *?st2*
using *assms istep-correct* **by** *blast*
then have *a2*: $\forall s.\ (s \rightarrow_S istep\ tm\ ?st1) \longrightarrow (s \rightarrow_C (s2,\ l2,\ r2))$
using *istate-eq-implies-config-eq*[**where** *c*=(*s2*, *l2*, *r2*)] **by** *presburger*

from *a1 a2* **show** *?thesis*
— We have to define the variables for conseq manually here, otherwise we get a timeout.
using *tm-imp-step-correct-aux conseq*[**where** *c*=*tm-imp-step tm*
and $P'=\lambda s.\ s \rightarrow_C (s1,\ l1,\ r1)$
and $P=\lambda s.\ s \rightarrow_S ?st1$
and $Q=\lambda s.\ s \rightarrow_S istep\ tm\ ?st1$
and $Q'=\lambda s.\ s \rightarrow_C (s2,\ l2,\ r2)$]
by *presburger*
qed

lemma *tm-imp-step-correct′*:
assumes *c1* $\models\langle tm\rangle=$ *c2*
shows $\vdash_t$ $\{\lambda s.\ s \rightarrow_C c1\}$ *tm-imp-step tm* $\{\lambda s.\ s \rightarrow_C c2\}$
proof −
obtain *s1 l1 r1* **where** *c1-def*: *c1* = (*s1*, *l1*, *r1*)
using *prod-cases3*[*of c1 thesis*] **by** *simp*
obtain *s2 l2 r2* **where** *c2-def*: *c2* = (*s2*, *l2*, *r2*)

using *prod-cases3*[*of c2 thesis*] **by** *simp*
from *assms c1-def c2-def* **show** *?thesis*
using *tm-imp-step-correct* **by** *simp*
qed

3.5 Repeated-Step Program

After constructing a single-step program, we now want to construct and verify a program, that continuously executes steps until it reaches a final state.

Note, that since we are using a Hoare logic for total correctness, verifying it means also proving its termination. This will make proofs later a bit tricky, since:

1. Termination of Turing-Machines is inherently undecidable, and thus will also be undecidable for our constructed program on arbitrary inputs.
2. Proof of Termination involve having a measurable number, that consistently decreases with each iteration. However, our state offers no such number at first glance.

We will solve this problem, by assuming that the TM-machine will also terminate. If it wouldn't, we can't pose any meaningful statements of our program, since it also wouldn't terminate. Given the assumed termination, we can then construct a number of steps required until a final state is reached. We will then use this number as our measurement of termination.

definition *tm-imp-steps* :: *tprog0* ⇒ *com* **where**
tm-imp-steps p = *WHILE* (*neq* (*N 0*) (*V vnS*)) *DO tm-imp-step p*

This lemma will help us determine a measurable number n, that specifies the remaining amount of steps required until a final state is reached, under the assumption that at *some point* a final state is reached.

lemma *tm-remaining-steps*:
assumes *is-final c2* **and** *c1* ⊨⟨*tm*⟩=* *c2*
obtains *n* **where** *steps0 c1 tm n* = *c2*
using *assms tm-steps0-rel-iff-steps0* **by** *blast*

— If we are in a final state, executing a step yields the same configuration.
lemma *step0-final*:
assumes *is-final c*
shows *step0 c tm* = *c*
using *assms is-final.elims*(*2*) **by** *fastforce*

— Same as above, but for multiple steps.
lemma *steps0-final*:
assumes *is-final c*
shows *steps0 c tm n* = *c*
by (*induction n*) (*use assms step0-final* **in** *simp*)+

Turing-Machines are deterministic in our model, which means whenver a TM reaches a final state from the same starting point, its final configuration will be determined:

lemma *tm-final-determined*:
assumes *c* ⊨⟨*tm*⟩=* *c1* **and** *c* ⊨⟨*tm*⟩=* *c2* **and** *is-final c1* **and** *is-final c2*

shows *c1* = *c2*
proof –
obtain *n1* **where** *n1-def*: *steps0 c tm n1* = *c1*
using *assms*(*1*) *tm-steps0-rel-iff-steps0* **by** *blast*
obtain *n2* **where** *n2-def*: *steps0 c tm n2* = *c2*
using *assms*(*2*) *tm-steps0-rel-iff-steps0* **by** *blast*

consider (*eq*) *n1* = *n2* | (*n1-first*) *n1* < *n2* | (*n2-first*) *n1* > *n2*
using *n1-def n2-def* **by** *linarith*
then show *?thesis* **proof** (*cases*)
case *eq*
then show *?thesis*
using *n1-def n2-def* **by** *simp*
next
case *n1-first*
then have *steps0* (*steps0 c tm n1*) *tm* (*n2*−*n1*) = *c2*
by (*metis le-add-diff-inverse n2-def order-less-imp-le steps-add*)
then have *steps0 c1 tm* (*n2*−*n1*) = *c2*
using *n1-def* **by** *blast*
then show *?thesis*
using *assms steps0-final* **by** *simp*
next
case *n2-first*
then have *steps0* (*steps0 c tm n2*) *tm* (*n1*−*n2*) = *c1*
by (*metis le-add-diff-inverse n1-def order-less-imp-le steps-add*)
then have *steps0 c2 tm* (*n1*−*n2*) = *c1*
using *n2-def* **by** *blast*
then show *?thesis*
using *assms steps0-final* **by** *simp*
qed
qed

3.5.1 Partial Correctness

lemma *total-implies-partial*:
assumes $\vdash_t$ {*P*} *c* {*Q*}
shows $\vdash$ {*P*} *c* {*Q*}
proof –
have $\models_t$ {*P*} *c* {*Q*}
using *hoaret-sound assms* **by** *simp*
then have $\models$ {*P*} *c* {*Q*}
unfolding *hoare-valid-def hoare-tvalid-def*
using *big-step-determ* **by** *blast*
then show *?thesis*
using *hoare-complete* **by** *simp*
qed

lemma *tm-imp-step-chain-total*: $\vdash_t$
$\{\lambda s.\ \exists c'.\ (s \rightarrow_C c') \wedge (c \models\langle tm\rangle=^* c')\}$
tm-imp-step tm

$\{\lambda s.\ \exists\, c'.\ (s \rightarrow_C\ c') \wedge (c \models\langle tm\rangle=^*\ c')\}$
proof −
  let *?P* = $\lambda s.\ \exists\, c'.\ (s \rightarrow_C\ c') \wedge (c \models\langle tm\rangle=^*\ c')$
  have $\bigwedge s.\ ?P\ s \Longrightarrow (\exists\, t.\ (tm\text{-}imp\text{-}step\ tm,\ s) \Rightarrow t \wedge ?P\ t)$
  proof −
    fix *s* :: *impstate*
    assume *?P s*
    then obtain *c1* **where** *c1-def*: $(s \rightarrow_C\ c1) \wedge (c \models\langle tm\rangle=^*\ c1)$
      by *blast*
    then obtain *c2* **where** *c2-def*: $(c1 \models\langle tm\rangle=\ c2)$
      by (*simp add*: *tm-step0-rel-def*)
    with *c1-def* **have** *c2-chain*: $(c \models\langle tm\rangle=^*\ c2)$
      using *tm-steps0-rel-def* **by** *force*

    have $\vdash_t \{\lambda s.\ s \rightarrow_C\ c1\}\ tm\text{-}imp\text{-}step\ tm\ \{\lambda s.\ s \rightarrow_C\ c2\}$
      using *tm-imp-step-correct′ c2-def* **by** *blast*
    then have $\models_t \{\lambda s.\ s \rightarrow_C\ c1\}\ tm\text{-}imp\text{-}step\ tm\ \{\lambda s.\ s \rightarrow_C\ c2\}$
      using *hoaret-sound* **by** *blast*
    then have $s \rightarrow_C\ c1 \Longrightarrow \exists\, t.\ (tm\text{-}imp\text{-}step\ tm,\ s) \Rightarrow t \wedge (t \rightarrow_C\ c2)$
      unfolding *hoare-tvalid-def* **by** *blast*
    then have $\exists\, t.\ (tm\text{-}imp\text{-}step\ tm,\ s) \Rightarrow t \wedge (t \rightarrow_C\ c2)$
      using *c1-def* **by** *blast*
    then show $\exists\, t.\ (tm\text{-}imp\text{-}step\ tm,\ s) \Rightarrow t \wedge ?P\ t$
      using *c2-chain* **by** *blast*
  qed
  then show *?thesis*
    using *hoaret-complete hoare-tvalid-def* **by** *presburger*
qed

lemma *tm-imp-steps-correct-partial*: $\vdash$
  $\{\lambda s.\ s \rightarrow_C\ c\}$
  tm-imp-steps tm
  $\{\lambda s.\ \exists\, c'.\ (s \rightarrow_C\ c') \wedge (c \models\langle tm\rangle=^*\ c') \wedge is\text{-}final\ c'\}$
proof −
  let *?b* = *neq* (*N 0*) (*V vnS*)
  let *?P* = $\lambda s.\ \exists\, c'.\ (s \rightarrow_C\ c') \wedge (c \models\langle tm\rangle=^*\ c')$

  have $\vdash$
    $\{\lambda s.\ ?P\ s \wedge bval\ ?b\ s\}$
    tm-imp-step tm
    $\{?P\}$
  proof −
    — Downgrade from Total Correctness to Partial Correctness.
    have *step*: $\vdash \{?P\}\ tm\text{-}imp\text{-}step\ tm\ \{?P\}$
      using *tm-imp-step-chain-total total-implies-partial* **by** *blast*
    show *?thesis* **by** (*rule partial-conseq′*) (*use step* **in** *blast*)+
  qed

  then have *loop*: $\vdash \{?P\}\ tm\text{-}imp\text{-}steps\ tm\ \{\lambda s.\ ?P\ s \wedge \neg\ bval\ ?b\ s\}$

unfolding *tm-imp-steps-def* **by** (*rule hoare.While*)

have *p*: $\bigwedge s.\ (s \to_C c) \Longrightarrow (?P\ s)$
using *tm-steps0-rel-def* **by** *blast*

have *q*: $\bigwedge s.\ (?P\ s \wedge \neg\ bval\ ?b\ s) \Longrightarrow (\exists\, c'.\ (s \to_C c') \wedge (c \models\langle tm\rangle=^* c') \wedge is\text{-}final\ c')$
proof −
fix $s :: impstate$
assume *assm*: $?P\ s \wedge \neg\ bval\ ?b\ s$
obtain c' **where** c'*-def*: $(s \to_C c') \wedge (c \models\langle tm\rangle=^* c')$
using *assm* **by** *blast*
then obtain $s'\ l'\ r'$ **where** c'*-split*: $c' = (s', l', r')$
using *prod-cases3* **by** *blast*
moreover have $s\ vnS = 0$
using *assm* **by** *simp*
ultimately have $s' = 0$
using *assm* c'*-def* **by** *simp*
then have *is-final* c'
using c'*-def* c'*-split* **by** *simp*
with c'*-def* **have** $(s \to_C c') \wedge (c \models\langle tm\rangle=^* c') \wedge is\text{-}final\ c'$
by *simp*
then show $\exists\, c'.\ (s \to_C c') \wedge (c \models\langle tm\rangle=^* c') \wedge is\text{-}final\ c'$
by (*rule exI*)
qed

show *?thesis* **by** (*rule partial-conseq'*) (*use loop p q* **in** *blast*)+
qed

3.5.2 Total Correctness

This lemma about `tm_imp_step` effectively says the same as our previous proof of correctness for it above. However, re-formulating the pre- and post-conditions makes it easier to integrate it into our proof of the WHILE-loop. To proof this variation of `tm_imp_step`, we choose a Big-Step approach.

lemma *tm-imp-step-chain'*:
assumes *is-final cf*
shows $\models_t$
$\{\lambda s.\ \exists\, c.\ s \to_C c \wedge \neg\ is\text{-}final\ c \wedge steps0\ c\ tm\ n =_C cf \wedge (cs \models\langle tm\rangle=^* c)\}$
tm-imp-step tm
$\{\lambda s.\ \exists\, c.\ s \to_C c \wedge n > 0 \wedge steps0\ c\ tm\ (n{-}1) =_C cf \wedge (cs \models\langle tm\rangle=^* c)\}$
unfolding *hoare-tvalid-def* **proof** (*standard*, *standard*)
fix $s :: impstate$
let $?P = \lambda c.\ s \to_C c$
$\wedge\ \neg\ is\text{-}final\ c$
$\wedge\ steps0\ c\ tm\ n =_C cf$
$\wedge\ (cs \models\langle tm\rangle=^* c)$
let $?Q = \lambda t\ c.\ t \to_C c \wedge n > 0 \wedge steps0\ c\ tm\ (n{-}1) =_C cf \wedge (cs \models\langle tm\rangle=^* c)$

assume $\exists\, c.\ ?P\ c$

then obtain c **where** *c-def*: *?P c*
  by *blast*

then obtain c' **where** c'*-def*: $c' =$ *step0 c tm*
  by *simp*
then have $c \models\langle tm\rangle= c'$
  by (*simp add*: *tm-step0-rel-def*)
then have $\vdash_t \{\lambda s.\ s \rightarrow_C c\}$ *tm-imp-step tm* $\{\lambda s.\ s \rightarrow_C c'\}$
  using *tm-imp-step-correct'* **by** *blast*
then have $\models_t \{\lambda s.\ s \rightarrow_C c\}$ *tm-imp-step tm* $\{\lambda s.\ s \rightarrow_C c'\}$
  using *hoaret-sound* **by** *blast*
then have $\forall s.\ s \rightarrow_C c \longrightarrow (\exists t.\ (tm\text{-}imp\text{-}step\ tm,s) \Rightarrow t \wedge t \rightarrow_C c')$
  using *hoare-tvalid-def* **by** *simp*
moreover have $s \rightarrow_C c$
  using *c-def* **by** *blast*
ultimately obtain t **where** *t-def*: $(tm\text{-}imp\text{-}step\ tm,\ s) \Rightarrow t \wedge (t \rightarrow_C c')$
  using *hoare-tvalid-def* **by** *presburger*

have *a1*: $(cs \models\langle tm\rangle=^* c')$
  using ‹$c \models\langle tm\rangle= c'$› *c-def tm-steps0-rel-def* **by** *force*

have *a2*: $n > 0$ **proof** (*cases* $n = 0$)
  case *True*
  then have *steps0 c tm n* $= c$
    by *simp*
  then have *steps0 c tm n* $=_C c$
    using *config-eq-refl'* **by** *simp*
  moreover have *steps0 c tm n* $=_C cf$
    using *c-def* **by** *blast*
  ultimately have $c =_C cf$
    using *config-eq-trans' config-eq-sym'* **by** *blast*
  moreover have $\neg$ *is-final c* $\wedge$ *is-final cf*
    using *c-def assms* **by** *blast*
  ultimately have *False*
    using *config-eq-det-is-final'* **by** *simp*
  then show *?thesis* **by** *simp*
next
  case *False*
  then show *?thesis* **by** *simp*
qed

obtain cf' **where** cf'*-def*: *steps0 c tm n* $= cf' \wedge cf' =_C cf$
  using *c-def* **by** *blast*
then have *steps0* c' *tm* $(n-1) = cf'$
  using *a2* c'*-def* **by** (*metis Suc-diff-1 steps.simps*(*2*))
with cf'*-def* **have** *a3*: *steps0* c' *tm* $(n-1) =_C cf$
  by *simp*

have $(tm\text{-}imp\text{-}step\ tm,\ s) \Rightarrow t \wedge\ ?Q\ t\ c'$

    using *t-def c′-def a1 a2 a3* **by** *blast*
  then show $\exists\, t.\ (\textit{tm-imp-step tm},\ s) \Rightarrow\ t \wedge (\exists\, c.\ ?Q\ t\ c)$
    by *blast*
qed

lemma *tm-imp-step-chain*:
  assumes *is-final cf*
  shows $\vdash_t$
    $\{\lambda s.\ \exists\, c.\ s \rightarrow_C c \wedge \neg\ \textit{is-final}\ c \wedge \textit{steps0}\ c\ tm\ n =_C cf \wedge (cs \models\langle tm\rangle{=}^*\ c)\}$
    tm-imp-step tm
    $\{\lambda s.\ \exists\, c.\ s \rightarrow_C c \wedge n > 0 \wedge \textit{steps0}\ c\ tm\ (n{-}1) =_C cf \wedge (cs \models\langle tm\rangle{=}^*\ c)\}$
  using *assms hoaret-complete tm-imp-step-chain′* **by** *blast*

Finally, we show that our `tm_imp_steps` program works as expected.

lemma *tm-imp-steps-correct-total*:
  assumes *is-final cf* **and** $cs \models\langle tm\rangle{=}^*\ cf$
  shows $\vdash_t \{\lambda s.\ s \rightarrow_C cs\}$ *tm-imp-steps tm* $\{\lambda s.\ s \rightarrow_C cf\}$
proof −
  let $?b = \textit{neq}\ (N\ 0)\ (V\ \textit{vnS})$
  let $?P = \lambda s.\ \exists\, c'.\ (cs \models\langle tm\rangle{=}^*\ c') \wedge s \rightarrow_C c'$
  let $?T = \lambda s\ n.\ \exists\, c'.\ s \rightarrow_C c' \wedge \textit{steps0}\ c'\ tm\ n =_C cf$

  have *step*: $\bigwedge n.\ \vdash_t$
    $\{\lambda s.\ ?P\ s \wedge \textit{bval}\ ?b\ s \wedge\ ?T\ s\ n\}$
    tm-imp-step tm
    $\{\lambda s.\ ?P\ s \wedge (\exists\, n'{<}n.\ ?T\ s\ n')\}$
  proof −
    fix $n :: \textit{nat}$

    let $?P' = \lambda s.\ \exists\, c.\ s \rightarrow_C c \wedge \neg\ \textit{is-final}\ c \wedge \textit{steps0}\ c\ tm\ n =_C cf \wedge (cs \models\langle tm\rangle{=}^*\ c)$
    let $?Q' = \lambda s.\ \exists\, c.\ s \rightarrow_C c \wedge n > 0 \wedge \textit{steps0}\ c\ tm\ (n{-}1) =_C cf \wedge (cs \models\langle tm\rangle{=}^*\ c)$

    have *p*: $\bigwedge s.\ (?P\ s \wedge \textit{bval}\ ?b\ s \wedge\ ?T\ s\ n) \Longrightarrow (?P'\ s)$
    proof −
      fix $s :: \textit{impstate}$
      assume *assm*: $?P\ s \wedge \textit{bval}\ ?b\ s \wedge\ ?T\ s\ n$

      obtain *c* **where** *a1*: $s \rightarrow_C c \wedge (cs \models\langle tm\rangle{=}^*\ c)$
        using *assm* **by** *blast*
      with *assm* **have** $\exists\, c'.\ c =_C c' \wedge \textit{steps0}\ c'\ tm\ n =_C cf$
        using *impstate-to-config-inv-det′* **by** *blast*
      then obtain *c′* **where** $c =_C c' \wedge \textit{steps0}\ c'\ tm\ n =_C cf$
        by *blast*
      then have *a2*: $\textit{steps0}\ c\ tm\ n =_C cf$
        using *config-eq-trans′ config-eq-steps0* **by** *blast*

      have $s\ \textit{vnS} \neq 0$
        using *assm* **by** *force*
      moreover obtain *st l r* **where** $c = (st,\ l,\ r)$
        using *prod-cases3* **by** *blast*

ultimately have *a3*: $\neg$ *is-final c*
using *a1* **by** *force*

from *a1 a2 a3* **show** *?P′ s* **by** *blast*
qed

have *q*: $\bigwedge s.\ (?Q'\ s) \Longrightarrow\ ?P\ s \wedge (\exists n'<n.\ ?T\ s\ n')$
proof −
fix *s* :: *impstate*
assume *assm*: *?Q′ s*

have *a1*: *?P s*
using *assm* **by** *blast*

have $n > 0 \wedge\ ?T\ s\ (n-1)$
using *assm* **by** *blast*
then have *a2*: $\exists n'<n.\ ?T\ s\ n'$
by (*cases n*, *simp*, *auto*)

from *a1 a2* **show** $?P\ s \wedge (\exists n'<n.\ ?T\ s\ n')$ **by** *blast*
qed

show $\vdash_t$
$\{\lambda s.\ ?P\ s \wedge bval\ ?b\ s \wedge\ ?T\ s\ n\}$
tm-imp-step tm
$\{\lambda s.\ ?P\ s \wedge (\exists n'<n.\ ?T\ s\ n')\}$
by (*rule conseq′*, *use tm-imp-step-chain*[**where** *n*=*n*] *assms*(*1*) **in** *fast*; *use p q* **in** *blast*)
qed

have *loop*: $\vdash_t$
$\{\lambda s.\ ?P\ s \wedge (\exists n.\ ?T\ s\ n)\}$
tm-imp-steps tm
$\{\lambda s.\ ?P\ s \wedge \neg\ bval\ ?b\ s\}$
unfolding *tm-imp-steps-def*
by (*rule While*, *use step* **in** *simp*)

have *p*: $\bigwedge s.\ (s \rightarrow_C cs) \Longrightarrow (?P\ s \wedge (\exists n.\ ?T\ s\ n))$
proof −
fix *s* :: *impstate*
assume *assm*: $s \rightarrow_C cs$

have $(cs \models\langle tm\rangle=^* cs) \wedge s \rightarrow_C cs$
using *assm* **by** (*simp add*: *tm-steps0-rel-def*)
then have *a1*: $\exists c'.\ (cs \models\langle tm\rangle=^* c') \wedge s \rightarrow_C c'$
by *blast*

obtain *n* **where** $steps0\ cs\ tm\ n =_C cf$
using *assms tm-remaining-steps config-eq-refl′* **by** *blast*
then have *a2*: $(\exists n.\ ?T\ s\ n)$

using *assm* **by** *blast*

from *a1 a2* **show** $?P\ s \wedge (\exists\, n.\ ?T\ s\ n)$ **by** *blast*
qed

have *q*: $\bigwedge s.\ (?P\ s \wedge \neg\ bval\ ?b\ s) \Longrightarrow (s \rightarrow_C cf)$
proof −
fix *s* :: *impstate*
assume *assm*: $?P\ s \wedge \neg\ bval\ ?b\ s$

obtain *c* **where** *c-def*: $(cs \models\langle tm\rangle{=}^*\ c) \wedge s \rightarrow_C c$
using *assm* **by** *blast*
then obtain *st l r* **where** $c = (st,\ l,\ r)$
using *prod-cases3* **by** *blast*
with *assm* **have** *is-final c*
using *c-def* **by** *simp*
with *c-def* **show** $s \rightarrow_C cf$
using *assms tm-final-determined* **by** *blast*
qed

show *?thesis* **by** (*rule conseq′*, *use loop* **in** *fast*; *use p q* **in** *fast*)
qed

3.6 IMP is Turing-Complete

We can now prove the main theorem of this project: that IMP is Turing-Complete. In words, we show that for every possible Turing-Machine *tm*, there exists an IMP-program *p*, that fulfils the following two properties:

1. For every configuration *c*, if *p* is started in a state that represents *c*, when *p* terminates it does so in state that represents a configuration c', where *tm* will also reach c' if started in *c* and which is a final-configuration, meaning the *tm* would also halt in exactly this configuration.

2. For every two configurations c_1 and c_2, where c_2 is a final-configuration, and *tm* would also reach (and halt) in c_2 if started in c_1, then our program, if started in a state that represents c_1, will terminate and its final state will represent c_2.

This shows, that when our constructed program terminates, it does so in a state we expect. Furthermore, this also shows that our constructed program always terminates, when we expect it to.

theorem *IMP-is-TuringComplete*:
fixes *tm* :: *tprog0*
obtains *p* **where** $\bigwedge c.\ \vdash \{\lambda s.\ s \rightarrow_C c\}\ p\ \{\lambda s.\ \exists\, c'.\ (s \rightarrow_C c') \wedge (c \models\langle tm\rangle{=}^*\ c') \wedge is\text{-}final\ c'\}$
and $\bigwedge c1\ c1.\ (is\text{-}final\ c2 \wedge (c1 \models\langle tm\rangle{=}^*\ c2)) \Longrightarrow (\vdash_t \{\lambda s.\ s \rightarrow_C c1\}\ p\ \{\lambda s.\ s \rightarrow_C c2\})$
using *tm-imp-steps-correct-partial tm-imp-steps-correct-total* **by** *blast*

The attentive reader will have noticed, that the first statement is formulated logically a bit differently, than described in words above. Precisely, we actually only show, that there exists a configuration c', that is represented by the state *s*, and that is final and reachable from *c*, but not that is uniquely defined.

This problem again stems from the ambigiuous definition of tapes in the imported definition. However, we have shown earlier in lemma impstate_to_config_inv_det that all possible configurations that could occur must be semantically equivalent with respect to our equivalence relation $=_C$.

With this, we have shown that IMP is Turing-Complete. □

end — end-theory IMP_TuringComplete

4 Closing Remarks

4.1 Conclusion

We have shown that IMP, as introduced by Tobias Nipkow and Gerwin Klein in "Concrete Semantics"[2], is Turing-Complete, by constructing an IMP-program, that can simulate a Turing-Machine.

4.2 Future Work

A possible continuation of this project could be to show the Turing-Equivalence of IMP: that is, showing that IMP can also be simulated by a Turing-Machine. Although, the widely-accepted Church-Turing-thesis implies that IMP can not be more powerful than a Turing-Machine. [1]

Another interesting future work could be investigating how different language features change the powerfulnes of IMP. For example, what effects would adding non-determinism (e.g. by an infinite loop statement) have, or what boundaries can be imposed without violating the Turing-Completeness.

Furthermore, this project could also be refined, replacing some abbreviations with definitions, making some proofs more difficult or require more facts, but also speeding up other proofs significantly. Some more utility lemmas can also be introduced, making some facts more directly derivable. Last, but not least, the Hoare pre-conditions and post-conditions may be refined in various facts, so that facts about them may be used more easily by other proofs.

References

[1] B. J. Copeland. The church-turing thesis. 1997.

[2] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. http://concrete-semantics.org.

[3] C. E. Shannon. *A Universal Turing Machine with Two Internal States*, pages 157–166. Princeton University Press, Princeton, 1956.

[4] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937.

[5] J. Xu, X. Zhang, C. Urban, S. J. C. Joosten, and F. Regensburger. Universal turing machine. *Archive of Formal Proofs*, February 2019. https://isa-afp.org/entries/Universal_Turing_Machine.html, Formal proof development.