Appendix for Verified Programming of Turing Machines in Coq

Yannick Forster
Saarland University
Saarbrücken, Germany
forster@ps.uni-saarland.de

Fabian Kunze
Saarland University
Saarbrücken, Germany
kunze@ps.uni-saarland.de

Maximilian Wuttke
Saarland University
Saarbrücken, Germany
s8mawutt@stud.uni-saarland.de

1 Semantics of Turing machine

We now give the semantics for Turing machines we omitted before.

Definition 1.1 (Configuration). *A configuration of* $M : \mathsf{TM}^n_\Sigma$ *is a tuple* $c = (q, t)$*, where* $q : Q_M$ *and* $t : \mathsf{Tape}^n_\Sigma$*. We write* $\mathsf{Conf}_M := Q_M \times \mathsf{Tape}^n_\Sigma$ *for the type of configurations of* M*.*

The function tape_move : Move → Tape_Σ → Tape_Σ moves a tape in a direction.

Definition 1.2 (Tape movement).

```
tape_move L (leftof r rs)          := leftof r rs
tape_move R (leftof r rs)          := midtape nil r rs
tape_move L (midtape nil m rs)     := leftof m rs
tape_move R (midtape ls m nil)     := rightof m ls
tape_move L (midtape (l :: ls) m rs) := midtape ls l (m :: rs)
tape_move R (midtape ls m (r :: rs)) := midtape (m :: ls) r rs
tape_move L (rightof l ls)         := midtape ls l nil
tape_move R (rightof l ls)         := rightof l ls
tape_move _ (niltape)              := niltape
tape_move N t                      := t
```

Note that moving further right (or left) when that tape already is to the right (or left) of the symbols, does not change the tape.

The functions left, right : Tape → $\mathcal{L}(\Sigma)$ return the symbols to the left (or right) side of the head:

Definition 1.3 (left and right).

```
  left (niltape)          := nil
 right (niltape)          := nil
  left (leftof r rs)      := nil
 right (leftof r rs)      := r :: rs
  left (midtape ls m rs)  := ls
 right (midtape ls m rs)  := rs
  left (rightof l ls)     := l :: ls
 right (rightof l ls)     := nil
```

Now we can define the function tape_write : Tape_Σ → $O(\Sigma)$ → Tape_Σ, that writes an optional symbol to a tape. When we write $\lfloor a \rfloor$, we get a midtape, where the left and right symbols remain unchanged and a is now in the middle. For $\emptyset$, the tape remains unchanged. Note that there is no way to decrease the number of symbols on a tape or to write 'blank' symbols.

Definition 1.4 (tape_write).

```
tape_write t ∅      := t
tape_write t ⌊a⌋    := midtape (left t) a (right t)
```

To define the function step : Conf → Conf, we need to know the symbols on the tapes. Therefore, we define a function current : Tape_Σ → $O(\Sigma)$. It returns $\emptyset$ if the head is not under a symbol, and $\lfloor m \rfloor$ if the head is under the symbol m.

Definition 1.5 (current). *The function* current : Tape_Σ → $O(\Sigma)$ *is defined by*

```
current (midtape ls m rs) := ⌊m⌋
current _                 := ∅
```

We can now define the step function step : Conf → Conf. First, the machine reads all the current symbols from the tapes. We apply this vector and the machine state to the transition function δ. Then, each tape writes the symbol and moves its head into the direction that δ yielded for it. The machine ends up in a new state q'.

Definition 1.6 (step).

```
doAct t (s, d) := tape_move d (tape_write t s)
step (q, t)    := let (q', act) := δ(q, map current t) in
                  (q', zipdoAct t act)
```

Here, zip : $\forall n\ A\ B\ C.(A \to B \to C) \to A^n \to B^n \to C^n$ is the zip-function on vectors.

To define the execution of a machine, we first define an abstract recursive function loop : $(A \to A) \to (A \to \mathbb{B}) \to A \to \mathbb{N} \to O(A)$ (for every $A : \mathbb{T}$):

Definition 1.7 (loop).

$$\mathsf{loop}\ f\ h\ a\ k := \begin{cases} \lfloor a \rfloor & h(a) \\ \emptyset & \neg h(a) \wedge k = 0 \\ \mathsf{loop}\ f\ h\ (f\ a)\ (k-1) & \neg h(a) \wedge k > 0 \end{cases}$$

We instantiate the abstract loop function and get a function loopM : Conf → $\mathbb{N}$ → O(Conf) that executes k steps of the machine:

CPP '20, January 20–21, 2020, New Orleans, LA, USA
2020.

Definition 1.8 (Machine execution). *For a fixed machine M, we define*

$$\begin{aligned}\mathsf{initConf}\ t &:= (t, \mathsf{start}_M)\\ \mathsf{haltConf}\ (t, q) &:= \mathit{halt}(q)\\ \mathsf{loopM}\ c\ k &:= \mathsf{loop\ step\ haltConf}\ c\ k\end{aligned}$$

We write $M(c) \triangleright^k c'$ for $\mathsf{loopM}\ c\ k = \lfloor c' \rfloor$ and $M(t) \triangleright^k c$ for $M(\mathsf{initConf}\ t) \triangleright^k c$.

1.1 Lifting operations

Concerning tape-lifts, we use the function select: For any m-vector I over indices $< n$, it implements a function selecting m components from any n-vector V:

Definition 1.9 (Vector selection). *Let $X : \mathbb{T}$, $m, n : \mathbb{N}$, $I : \mathbb{F}_n^m$, and $V : X^m$. Then* $\mathsf{select}\ I\ V : X^n$ *is defined by* $\mathsf{select}\ I\ V := (\lambda(j : \mathbb{F}_m).\ V[j])@I$.

This can now lift any realisation and running time relation by selecting the right subset of tapes from the larger, 'real' tapes, and plugging them in the smaller relation that gets lifted:

Definition 1.10 (Tape-Lift of Relations). *Let $I : \mathbb{F}_n^m$ and $R \subseteq \mathsf{Tape}^m \times (L \times \mathsf{Tape}^m)$ and $T \subseteq \mathsf{Tape}^m \times \mathbb{N}$, then we define*

$$\begin{aligned}\Uparrow_I R &:= \lambda t\ (\ell, t').\ R\ (\mathsf{select}\ I\ t)\ (\ell, \mathsf{select}\ I\ t')\\ \Uparrow_I T &:= \lambda t\ k.\ T\ (\mathsf{select}\ I\ t)\ k\end{aligned}$$

Concerning alphabet-lifts, we need to translate symbols backwards using retracts. Therefore, we need a default symbol d: for any symbol not covered by the retract.

Definition 1.11 (Alphabet translation). *For a retract $f : \Sigma \hookrightarrow \Gamma$ and default element $d : \Sigma$, the surject $f\ d : \Gamma \to \Sigma$ is defined by*

$$\mathsf{surject}\ f\ d\ \tau := \begin{cases}\sigma & \text{if } f^{-1}(\tau) = \lfloor \sigma \rfloor\\ d & \text{otherwise}\end{cases}$$

The alphabet-lift of the relations then needs to translate the tape content from the larger, 'real' alphabet to the smaller alphabet used in the relation to be lifted. It does so by mapping surject over the tapes.

Definition 1.12 (Alphabet-Lift of Relations). *Let $f : \Sigma \hookrightarrow \Gamma$ and $d : \Sigma$ and $R \subseteq \mathsf{Tape}^m \times (L \times \mathsf{Tape}^m)$ and $T \subseteq \mathsf{Tape}^m \times \mathbb{N}$, then we define*

$$\begin{aligned}\Uparrow_{(f,d)} R &:= \lambda t\ (\ell, t').\ R\ ((\mathsf{surject}\ f\ d)@t)\ (l, (\mathsf{surject}\ f\ d)@t')\\ \Uparrow_{(f,d)} T &:= \lambda t\ k.\ T\ ((\mathsf{surject}\ f\ d)@t)\ k\end{aligned}$$

2 Memorising While Loop

To ease the implementation of the translation from multi tape to single tape machines we use a machine implementing a generalisation of both Switch and While. $\mathsf{MemWhile}\ f\ x_0$ allows memorising a label between loop iterations, without writing it to a tape: To pass values of the finite type X between iterations and ultimately return labels over Y, a family $f : X \to \mathsf{TM}_\Sigma^n(X + Y)$ of machines labelled over the direct sum is required. The resulting machine is constructed in such a way that it first executes the machine $f\ x_0$, returning labels in $X + Y$. If this results in a label $x : X$, the execution loops, and executes $f\ x$ next. This is repeated until a label $y : Y$ is produced, which denotes that the loop has finished and is the label the combinator returns. We omit the precise construction, realisation and running time relation this machine satisfies for brevity; they are a combination of the respective relations from Switch and While.