Verified Programming of Turing Machines in Coq

Yannick Forster  Fabian Kunze  Maximilian Wuttke

Saarland University, Saarland Informatics Campus

January 20
CPP 2020
Why Turing Machines?

Traditional basis of theory of computation & complexity:

- Simple model of computation
- Straightforward notion of time & space consumption

Finite Control

Forster, Kunze, Wuttke
Verified Programming of TMs in Coq
January 20 CPP 2020
Turing machines are difficult

Hard to construct and verify:

- Not inherently compositional:
  - States & transition function
  - Number of tapes
  - Different alphabets
- Data not structured: strings over alphabet

Therefore no mechanisation of complexity-theoretic results available like:

\[
P \subseteq NP \subseteq PSPACE \subseteq EXP \]

Cook-Levin theorem: SAT is NP complete

Forster, Kunze, Wuttke
Verified Programming of TMs in Coq
January 20 CPP 2020
Turing machines are difficult

Hard to construct and verify:

- Not inherently compositional:
  - States & transition function
  - Number of tapes
  - Different alphabets
- Data not structured: strings over alphabet

Therefore no mechanisation of complexity-theoretic results available like:

- $P \subseteq NP \subseteq PSPACE \subseteq EXP$
- Cook-Levin theorem: SAT is NP complete
- Time & space hierarchy theorem
Our vision: Complexity theory in call-by-value \(\lambda\)-Calculus L

Why use L?

- Tools for verification and running time analysis in L (Forster&Kunze (2019))

L induces same notion of computability:

- **Church-Turing thesis**: L and TMs can simulate each other

Complexity theory possible?

- **Invariance thesis** justifies this: L and TMs can simulate each other with polynomial time overhead and constant factor space overhead.
  
  - Proof: *The Weak Call-By-Value \(\lambda\)-Calculus is Reasonable for Both Time and Space*  
    
    POPL Wednesday 10:30, Forster&Kunze&Roth (2020) (non-mechanised)

Mechanisation of the POPL result will require resource-aware verification of TMs\(^1\)

---

\(^1\)Church-Turing already mechanised using the results of this paper
Related Work: Mechanised verification of Turing machines

Mechanised universal Turing machines (correctness & termination):
- In Isabelle/HOL, using a Hoare logic: Xu&Zhang&Urban (2013)

We extend Asperti&Ricciotti’s verification approach with time & space analysis, in Coq
Contribution

A framework to:

- Construct Turing machines
- Verify correctness & termination
- Verify/deduce time and space complexity

Case studies:

- Addition and multiplication of numbers
- Several operations on lists
- Turing machine interpreter (Universal TM)
- Multi-tape to single-tape translation
Constructing Machines

Example: 2 tapes, Copy symbol left of head on tape 0 to tape 1.

CopyLeft : TM^2_{Σ} :=
\[ \uparrow_{[0]}(\text{Move L}); \]
Switch (\[ \uparrow_{[0]} \text{Read} \])
\[ (\lambda f. \text{match } f \text{ with} \]
\[ [s] \Rightarrow \uparrow_{[1]}(\text{Write } s) \]
\[ | \emptyset \Rightarrow \text{Nop} ) \]

- Shallowly embedded language constructs machines
- Primitive machines: Move d, Read, Write s
- Control flow combinators: \( M_1; M_2, \text{Switch } M (\lambda f . M_f) \)
- Tape selection: \( \uparrow_I M \)
Approach: Abstraction Layers

0 Multi-tape Turing machines (definition and semantics)
1 Labelled Turing machines & specification predicates
2 Control-flow & lifting operators
3 Registers: tape contain encodable types ($\mathbb{N}$, lists, ... )
Layer 0: Bare Turing Machines

From Asperti & Ricciotti:

- $n$-tape Turing machines $M : \text{TM}_\Sigma^n$ over alphabet $\Sigma$.
- Semantics: evaluation in $k$ steps, $M(q, t) \triangleright^k (q', t')$
- $t : \text{Tape}_\Sigma$ is canonical, blank-free representation of tapes
  - Used space visible in resulting tape since no deallocation can happen
Layer 1: Labelled Machines & Relations

We only care *what* can be computed:

- Labelled machine $M : \text{TM}_\Sigma^n(F)$ over finite Type $F$ to hide state names:

$$M = (M' : \text{TM}_\Sigma^n, \text{lab} : Q_{M'} \rightarrow F)$$
Layer 1: Labelled Machines & Relations

We only care *what* can be computed:

- Labelled machine $M : \text{TM}_\Sigma^n(F)$ over finite Type $F$ to hide state names:

  $$M = (M' : \text{TM}_\Sigma^n, \text{lab} : Q_{M'} \rightarrow F)$$

- Realisation of $R : \text{Tape}_\Sigma^n \rightarrow (F \times \text{Tape}_\Sigma^n) \rightarrow \mathbb{P}$:

  $$M \models R := \forall t \ q \ t'. M(t) \triangleright (q, t') \rightarrow R \ t \ (\text{lab}_M \ q, t')$$

  Can include space consumption.
Layer 1: Labelled Machines & Relations

We only care what can be computed:

- Labelled machine $M : \text{TM}_\Sigma^n(F)$ over finite Type $F$ to hide state names:

  $$M = (M' : \text{TM}_\Sigma^n, \text{lab} : Q_{M'} \rightarrow F)$$

- Realisation of $R : \text{Tape}_\Sigma^n \rightarrow (F \times \text{Tape}_\Sigma^n) \rightarrow \mathbb{P}$:

  $$M \models R := \forall t \ q \ t'. \ M(t) \triangleright (q, t') \rightarrow R \ t \ (\text{lab}_M q, t')$$

  Can include space consumption.

- Termination in $T : \text{Tape}_\Sigma^n \rightarrow \mathbb{N} \rightarrow \text{Prop}$:

  $$M \downarrow T := \forall t \ k. \ T \ t \ k \rightarrow \exists c. \ M(t) \triangleright^k c$$

  Includes time analysis.
Layer 2: Basic Machines

The “assembly commands” of our (shallowly embedded) language:
Layer 2: Basic Machines

The “assembly commands” of our (shallowly embedded) language:

- Write symbol \( s \) to the tape:

\[
\text{Write } s \vdash^c \lambda t \; t'. \quad t'[0] = \text{tape\_write } s \; t[0]
\]
Layer 2: Basic Machines

The “assembly commands” of our (shallowly embedded) language:

- Write symbol $s$ to the tape:

  $$\text{Write } s \models^c \lambda \ t \ t'. \ t'[0] = \text{tape_write} \ s \ t[0]$$

- Read the current symbol:

  $$\text{Read } \models^c \lambda \ (\ell, t'). \ \ell = \text{current} \ (t[0]) \land t = t'$$
Layer 2: Basic Machines

The “assembly commands” of our (shallowly embedded) language:

- Write symbol $s$ to the tape:

  $\text{Write } s \vdash^c \lambda t \ t'. \ t'[0] = \text{tape_write } s \ t[0]$

- Read the current symbol:

  $\text{Read } \vdash^c \lambda (\ell, \ t'). \ \ell = \text{current } (t[0]) \land t = t'$

- Move $d$
Layer 2: Control Flow Combinators

\[
M_1 \models R_1 \quad M_2 \models R_2 \\
\frac{}{M_1; M_2 \models R_1 \circ R_2}
\]
Layer 2: Control Flow Combinators

\[
\begin{align*}
M_1 & \models R_1 & M_2 & \models R_2 \\
\frac{}{M_1; M_2 \models R_1 \circ R_2}
\end{align*}
\]

\[
\begin{align*}
M_1 & \models R_1 & M_1 & \Downarrow T_1 & M_2 & \Downarrow T_2 \\
\frac{}{M_1; M_2 \Downarrow (\lambda t\ k. \ \exists k_1 \ k_2. \ (1 + k_1 + k_2) \leq k \land T_1 t \ k_1 \land \forall t' \ \ell. \ R_1 t (\ell, t') \rightarrow T_2 t' \ k_2)}
\end{align*}
\]
Layer 2: Control Flow Combinators

\[
\frac{M_1 \models R_1 \quad M_2 \models R_2}{M_1; M_2 \models R_1 \circ R_2}
\]

\[
\frac{M_1 \models R_1 \quad M_1 \downarrow T_1 \quad M_2 \downarrow T_2}{M_1; M_2 \downarrow (\lambda t. k. \exists k_1 k_2. (1 + k_1 + k_2) \leq k \land T_1 t k_1 \land \forall t' \ell. R_1 t (\ell, t') \rightarrow T_2 t' k_2)}
\]

- Conditional: If \( M_1 \) Then \( M_2 \) Else \( M_3 \)
- Loop doWhile \( M : TM(F) \) (for \( M : TM(\text{option } F) \))
Layer 2: Liftings

- Combine 2-tape machine $M$ and 5-tape machine $N$:

$$\uparrow_{[2,4]} M; N$$

- Other lifting to change alphabet

- Realisation/termination: relations can be lifted as well.
Layer 3: Compound data in tapes

- Type $X$ encodable in alphabet $\Sigma$ means there exists injective function $\varepsilon : X \rightarrow \Sigma^*$
- Notion $t \simeq_k x$ means $t$ contains encoding of $x$ and $\leq k$ other used cells
- Encodable: $\mathbb{N}$, pairs, lists, inductive data types...
- Constructor and destructor machines, e.g. for $\mathbb{N}$:
  - ConstrO: Write 0 to tape
  - ConstrS: Increment number encoded on tape
  - CaseNat : TM($\mathbb{B}$): destruct number and return occurred constructor.
How to use all this?

Recall Example:

$\text{CopyLeft} : \text{TMS}_\Sigma \triangleq \uparrow_0 \text{Move L; Switch (}\uparrow_0 \text{Read)}(\lambda f. \text{match } f \text{ with } \ldots$

We want $\text{CopyLeft} \vdash \text{CopyLeftRel}$
How to use all this?

Recall Example:

\( \text{CopyLeft} : \text{TM}_\Sigma^2 := \uparrow[0] \text{Move L; Switch (\uparrow[0] \text{Read})(\lambda f. \text{match } f \text{ with } \ldots} \)

We want \( \text{CopyLeft} \models \text{CopyLeftRel} \)

Our framework and automation reduces all this to relational inclusion:

\[
\uparrow[0](\lambda t_0 \ t_1. \ t_1[0] = \text{tape}_\text{move L } t_0[0])
\circ (\lambda t_1 ((\ell' : \mathbb{1}), t_3).
\exists t_2 (\ell : \text{option}(\Sigma)). (\uparrow[0](\lambda t_1 (\ell, t_2). \ell = \text{current}(t[0]) \land t_2 = t_1)) \ t_1 (\ell, t_2)
\land (\text{match } \ell \text{ with }
\begin{align*}
| s & \Rightarrow \text{LiftTapes}(\lambda t_2 \ t_3. \ t_3[0] = \text{tape}_\text{write } s \ t_2[0])[1] \ t_2 \ t_3 \\
| \emptyset & \Rightarrow t_3 = t_2)
\end{align*}
\subseteq \text{CopyLeftRel}
\]
How to use all this?

Recall Example:

\[ \text{CopyLeft} : \text{TM}_{\Sigma}^2 := \uparrow[0] \text{Move L; Switch (}\uparrow[0] \text{Read})(\lambda f. \text{match } f \text{ with } \ldots) \]

We want \( \text{CopyLeft} \models \text{CopyLeftRel} \)

Our framework and automation reduces all this to relational inclusion:

\[ \uparrow[0](\lambda t_0. t_1[0] = \text{tape\_move L } t_0[0]) \]
\[ \circ (\lambda t_1 ((\ell' : \Sigma), t_3). \]
\[ \exists t_2 ((\ell : \text{option}(\Sigma)). (\uparrow[0](\lambda t_1 (\ell, t_2). \ell = \text{current}(t[0]) \land t_2 = t_1)) t_1 (\ell, t_2) \]
\[ \land (\text{match } \ell \text{ with} \]
\[ [s] \Rightarrow \text{LiftTapes}(\lambda t_2. t_3[0] = \text{tape\_write s } t_2[0])[1] t_2 t_3 \]
\[ |\emptyset \Rightarrow t_3 = t_2) \]
\[ \subseteq \text{CopyLeftRel} \]

And similar for termination/time.
Main improvements compared to Asperti&Ricciotti

- Explicit handling of time and space
- Explicit notion of encoded datatypes (Layer 3)
- Labelled machines
Case studies

- Addition and multiplication of numbers \(^2\)
- Several list operations
- Turing machine interpreter (Universal TM)
- Multi-tape to single-tape translation (without layer 3)

\(^2\) unary, but binary in files
Universal 1-tape Turing Machine

Theorem

There exists a universal Turing machine $\text{Univ}_\Sigma$ that, given an encoded single-tape machine $M$ over $\Sigma$ and an encoded input tape $t_M$, simulates $M$ on $t_M$ with polynomial time overhead and constant-factor space overhead.
Universal 1-tape Turing Machine

**Theorem**

*There exists a universal Turing machine* $\text{Univ}_\Sigma$ *that, given an encoded single-tape machine* $M$ *over* $\Sigma$ *and an encoded input tape* $t_M$, *simulates* $M$ *on* $t_M$ *with polynomial time overhead and constant-factor space overhead.*

Actually two theorems, e.g. for correctness/space:

$$\text{Univ}_\Sigma \models \lambda t \ t'.
\forall (M : \text{TM}_1^\Sigma) (t_M : \text{Tape}_\Sigma) (q : Q_M) (s_1 s_2 s_3 s_4 s_5 : \mathbb{N}).
\begin{align*}
t[0] &\simeq t_M \rightarrow t[1] \simeq_{s_1} \delta_M \rightarrow t[2] \simeq_{s_2} q \rightarrow \\
\text{isVoid}_{s_3}(t[3]) &\rightarrow \text{isVoid}_{s_4}(t[4]) \rightarrow \text{isVoid}_{s_5}(t[5]) \rightarrow \\
\exists (t'_M : \text{Tape}_\Sigma) (q' : Q_M). M(q, t_M) \rhd (q', t'_M) \land \\
t'[0] &\simeq t'_M \land t'[1] \simeq_{s_1} \delta_M \land t[2] \simeq (2 + |Q_M| + \max c_M s_2) q' \land \\
\text{isVoid}_{\max c_M s_3}(t'[3]) &\land \text{isVoid}_{\max c_M s_4}(t'[4]) \land \text{isVoid}_{\max c_M s_5}(t'[5]), \text{ where } c_M := |\varepsilon(\delta_M)| + 1
\end{align*}$$

And similarly for time.
Mechanisation in Coq

Useful features:

- Tactics and Ltac (proof state unmanageable otherwise)
- Existential variables
- smpl plugin\(^3\) by Sigurd Schneider:
  - Automated forward reasoning and simplification
  - Extendable by hints
  - Should be part of Coq’s `auto`

Lessons learned:

- We should have used mathcomp for finite types.
- Inductive tape `(Sigma:finType) := ...`

---

\(^3\)https://github.com/sigurdschneider/smpl
Mechanisation in Coq (2)

- Paper hyperlinks to Coq development
- 5 min compile time
- Total: 19,100 loc
- Universal machine: 1844 lines
Lesson learned

The project started October 2016. Now, the verifications are possible, yet very tedious.
Lesson learned

The project started October 2016. Now, the verifications are possible, yet very tedious.

Formalising\(^4\) complexity theoretical results,
(like Cook-Levin theorem, \(P \subseteq NP \subseteq PSPACE \subseteq \text{EXP} \ldots\))
with Turing machines is inherently infeasable.

\(^4\)not only mechanising
The project started October 2016. Now, the verifications are possible, yet very tedious.

Formalising\(^4\) complexity theoretical results, (like Cook-Levin theorem, \(P \subseteq NP \subseteq PSPACE \subseteq EXP \ldots\)) with Turing machines is inherently infeasable.

\(^4\)not only mechanising
Conclusion

Future Work:

- Mechanise call-by-value $\lambda$-calculus interpreters needed for invariance thesis:
  
  Yannick Forster & Fabian Kunze & Marc Roth;
  *The Weak Call-By-Value $\lambda$-Calculus is Reasonable for Both Time and Space;*
  POPL 2020, Wednesday 10:30

- Investigate complexity theory in the call-by-value $\lambda$-calculus:
  
  - Time hierarchy
  - Cook-Levin theorem (SAT is NP-complete)

Framework used in the library for undecidable problems to reduce halting problem from L to TMs: CoqPL(Saturday 16:05), github.com/uds-psl/coq-library-undecidability
Conclusion

Future Work:

- Mechanise call-by-value $\lambda$-calculus interpreters needed for invariance thesis:
  
  Yannick Forster & Fabian Kunze & Marc Roth;  
  *The Weak Call-By-Value $\lambda$-Calculus is Reasonable for Both Time and Space;*  
  POPL 2020, Wednesday 10:30

- Investigate complexity theory in the call-by-value $\lambda$-calculus:
  
  - Time hierarchy
  - Cook-Levin theorem (SAT is NP-complete)

Framework used in the library for undecidable problems to reduce halting problem from L to TMs: CoqPL(Saturday 16:05), github.com/uds-psl/coq-library-undecidability

Thanks
<table>
<thead>
<tr>
<th>Component</th>
<th>Spec</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libraries (finite types, retractions, etc.)</td>
<td>2157</td>
<td>2716</td>
</tr>
<tr>
<td>$L_0$ &amp; $L_1$</td>
<td>Semantics of (labelled) TMs</td>
<td>644</td>
</tr>
<tr>
<td></td>
<td>Primitive Machines</td>
<td>178</td>
</tr>
<tr>
<td>$L_2$</td>
<td>Lifts</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>Combinators</td>
<td>561</td>
</tr>
<tr>
<td></td>
<td>Simple machines</td>
<td>598</td>
</tr>
<tr>
<td>$L_3$</td>
<td>Encodable types</td>
<td>282</td>
</tr>
<tr>
<td></td>
<td>Tape-containment</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>(De-) constructors</td>
<td>501</td>
</tr>
<tr>
<td></td>
<td>CopyValue, Reset, Compare</td>
<td>538</td>
</tr>
<tr>
<td></td>
<td>Refinement on alphabet-lift</td>
<td>139</td>
</tr>
<tr>
<td>Ltac automation (for $L_2$ and $L_3$)</td>
<td></td>
<td>156</td>
</tr>
<tr>
<td>Complexity</td>
<td>Time (infrastructure and machines)</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>Space (infrastructure and machines)</td>
<td>259</td>
</tr>
<tr>
<td>Add and Mult</td>
<td></td>
<td>222</td>
</tr>
<tr>
<td>$M_1$</td>
<td>Encoding</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td>783</td>
</tr>
<tr>
<td></td>
<td>Time bounds</td>
<td>236</td>
</tr>
<tr>
<td>Univ</td>
<td>Lookup in association list</td>
<td>169</td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td>324</td>
</tr>
<tr>
<td></td>
<td>Time bounds</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>Space bounds</td>
<td>240</td>
</tr>
<tr>
<td>MU (single-tape Univ for multi-tape machines)</td>
<td></td>
<td>408</td>
</tr>
<tr>
<td>Total (excl. libraries)</td>
<td></td>
<td>7327</td>
</tr>
</tbody>
</table>

5 minutes @ Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz machine.
Semantics TM

\[ M : \text{TM}_\Sigma^n := (Q : \text{finType}, \]
\[ s : Q, \]
\[ h : Q \rightarrow \mathbb{B}, \]
\[ \delta : Q \times (\text{option}(\Sigma))^n \rightarrow Q \times (\text{option}(\Sigma) \times \text{Move})^n ) \]