The Weak Call-By-Value λ-Calculus is Reasonable for Both Time and Space

Yannick Forster, Fabian Kunze, Marc Roth

POPL 2020
January 22

SAARLAND UNIVERSITY
COMPUTER SCIENCE

CLUSTER OF EXCELLENCE

SIC Saarland Informatics Campus

The strong invariance thesis

Slot and Van Emde Boas (1984):

Reasonable machines
can simulate each other with a polynomial overhead in time and a constant-factor overhead in space

The strong invariance thesis

Slot and Van Emde Boas (1984):

Turing machines and RAM machines
can simulate each other with a polynomial overhead in time and a constant-factor overhead in space

The strong invariance thesis

This paper:

Turing machines and the weak call-by-value λ-calculus
can simulate each other with a polynomial overhead in time and a constant-factor overhead in space

The strong invariance thesis

This paper:

Turing machines and the weak call-by-value λ-calculus
can simulate each other with a polynomial overhead in time and a constant-factor overhead in space
with respect to the natural measures,
i.e. β-steps and size of largest term in the computation

The strong invariance thesis

This paper:

Turing machines and the weak call-by-value λ-calculus
can simulate each other with a polynomial overhead in time and a constant-factor overhead in space
with respect to the natural measures,
i.e. β-steps and size of largest term in the computation

Standard complexity classes like P, NP, PSPACE, EXP
can be defined in terms of the λ-calculus[1]

[1] our result does not cover sublinear space or time

The Weak Call-by-Value λ-calculus L (Forster & Smolka, 2017)

$$s, t, u, v \;:\; \mathsf{Ter} \;::=\; n \mid st \mid \lambda s \qquad \text{where } n : \mathbb{N}.$$

- Weak evaluation: don't reduce below abstractions
- Call-by-value evaluation: reduce arguments first
- Deterministic

Define (tree-)size of terms:

$$\|n\| := 1 + n \qquad \|\lambda s\| := 1 + \|s\| \qquad \|st\| := 1 + \|s\| + \|t\|$$

For $s = s_0 \succ s_1 \succ \cdots \succ s_k = \lambda x.u$ define

- Time measure of s: k
- Space measure of s: $\max_{i=0}^{k} \|s_i\|$

The Weak Call-by-Value λ-calculus L (Forster & Smolka, 2017)

$$s, t, u, v \;:\; \mathsf{Ter} \;::=\; n \mid st \mid \lambda s \qquad \text{where } n : \mathbb{N}.$$

- Weak evaluation: don't reduce below abstractions
- Call-by-value evaluation: reduce arguments first
- Deterministic

Define (tree-)size of terms:

$$\|n\| := 1 + n \qquad \|\lambda s\| := 1 + \|s\| \qquad \|st\| := 1 + \|s\| + \|t\|$$

For $s = s_0 \succ s_1 \succ \cdots \succ s_k = \lambda x.u$ define

- Time measure of s: k
- Space measure of s: $\max_{i=0}^{k} \|s_i\|$

Measures are mapping terms to numbers!

The easy direction

Theorem

The weak call-by-value λ-calculus can simulate Turing machines with a polynomial overhead in time and a constant-factor overhead in space.

Follows by Accattoli / Dal Lago '17,
result for time mechanised in Forster / Kunze '19.

What's known for the harder direction?

1996: Lawall and Mairson: "total ink used" and "maximum ink used" are reasonable measures for the full λ-calculus

2008: Dal Lago and Martini: β-steps and accounting for the size of β-redexes are a reasonable time measure for the weak call-by-value λ-calculus

2016: Accattoli and Dal Lago: (leftmost-outermost) β-steps are a reasonable time measure for the full λ-calculus

Why isn't that enough?

To allow for e.g. mechanised complexity theoretic results we want

- **compositional** measures
- *for a* **compositional model**
- *for* **both** *time and space*

Why isn't that enough?

To allow for e.g. mechanised complexity theoretic results we want

- **compositional** measures

  "total ink used" for time by Lawall/Mairson is not enough

- *for a* **compositional model**

- *for* **both** *time and space*

Why isn't that enough?

To allow for e.g. mechanised complexity theoretic results we want

- **compositional** measures

  "total ink used" for time by Lawall/Mairson is not enough

- *for a* **compositional model**

  Turing machines are not enough

- *for* **both** *time and space*

Why isn't that enough?

To allow for e.g. mechanised complexity theoretic results we want

- **compositional** measures

  "total ink used" for time by Lawall/Mairson is not enough

- *for a* **compositional model**

  Turing machines are not enough

- *for* **both** *time and space*

  time complexity results by Accattoli/Dal Lago/Martini are not enough

Wait a moment!

There's a λ-term s_E with

$$\forall n : \mathbb{N}.\ s_E\ \overline{n} \succ^* \lambda xy.x$$

in $\mathcal{O}(n)$ steps but with $\mathcal{O}(2^n)$ space.

Wait a moment!

There's a λ-term s_E with

$$\forall n : \mathbb{N}.\ s_E\ \overline{n} \succ^* \lambda xy.x$$

in $\mathcal{O}(n)$ steps but with $\mathcal{O}(2^n)$ space.

But P $\subseteq$ PSPACE?!

$\mathrm{P} \subseteq \mathrm{PSPACE}$

Let M be a machine computing a function $\mathbb{N} \to \mathbb{B}$ and $x : \mathbb{N}$

Theorem (for TMs, free)

$$\mathcal{S}(M, x) \leq \mathcal{T}(M, x)$$

P $\subseteq$ PSPACE

Let M be a machine computing a function $\mathbb{N} \to \mathbb{B}$ and $x : \mathbb{N}$

Theorem (for TMs, free)

$$\mathcal{S}(M, x) \leq \mathcal{T}(M, x)$$

Theorem (for RAM machines, Slot & van Emde Boas, hard)

$$\mathcal{S}(M, x) \leq p(\mathcal{T}(M, x)) \qquad \textit{for a polynomial } p$$

$\mathsf{P} \subseteq \mathsf{PSPACE}$

Let M be a machine computing a function $\mathbb{N} \to \mathbb{B}$ and $x : \mathbb{N}$

Theorem (for TMs, free)

$$\mathcal{S}(M,x) \leq \mathcal{T}(M,x)$$

Theorem (for RAM machines, Slot & van Emde Boas, hard)

$$\mathcal{S}(M,x) \leq p(\mathcal{T}(M,x)) \qquad \textit{for a polynomial } p$$

Theorem (for L, follows from our result)

$$\exists M'.\ \ M' \textit{ is ext. equiv. to } M \textit{ and}$$
$$\mathcal{S}(M',x) \leq p_1(\mathcal{T}(M',x)) \leq p_2(\mathcal{T}(M,x))$$
$$\textit{for polynomials } p_1, p_2$$

Terms can exhibit size explosion,
but decision functions can be optimised
to not explode.

Theorem (Accattoli & Dal Lago, 2016)

There is an algorithm which takes as input a λ-term t and which, in time polynomial in the number of left-most outermost β-steps of t and the size of t

outputs an LSC term u such that the unfolding of u is the normal form of t.

Theorem (This paper, 2020)

There is an algorithm which takes as input a **closed L-term** *t and which,*

Theorem (Accattoli & Dal Lago, 2016)

There is an algorithm which takes as input a λ-term t and which, in time polynomial in the number of left-most outermost β-steps of t and the size of t

outputs an LSC term u such that the unfolding of u is the normal form of t.

Theorem (This paper, 2020)

There is an algorithm which takes as input a **closed L-term** *t and which, in time polynomial in the number of β-steps of t and the size of t*

Theorem (Accattoli & Dal Lago, 2016)

There is an algorithm which takes as input a λ-term t and which, in time polynomial in the number of left-most outermost β-steps of t and the size of t

outputs an LSC term u such that the unfolding of u is the normal form of t.

Theorem (This paper, 2020)

There is an algorithm which takes as input a **closed L-term** *t and which, in time polynomial in the number of β-steps of t and the size of t* **and space linear in the size of the largest term in the reduction**

Theorem (Accattoli & Dal Lago, 2016)

There is an algorithm which takes as input a λ-term t and which, in time polynomial in the number of left-most outermost β-steps of t and the size of t

outputs an LSC term u such that the unfolding of u is the normal form of t.

Theorem (This paper, 2020)

There is an algorithm which takes as input a **closed L-term** *t and which, in time polynomial in the number of β-steps of t and the size of t* **and space linear in the size of the largest term in the reduction** *outputs a* **heap containing a term** *u such that the unfolding of u is the normal form of t.*

Theorem (Accattoli & Dal Lago, 2016)	Theorem (This paper, 2020)
There is an algorithm which takes as input a λ-term t and which, in time polynomial in the number of left-most outermost β-steps of t and the size of t outputs an LSC term u such that the unfolding of u is the normal form of t.	*There is an algorithm which takes as input a* **closed L-term** *t and which, in time polynomial in the number of β-steps of t and the size of t* **and space linear in the size of the largest term in the reduction** *outputs a* **heap containing a term** *u such that the unfolding of u is the normal form of t.*

Unfolding takes time polynomial in number of β-steps and size of result.

For decision functions, evaluation is polynomial and
P, NP, PSPACE, EXP can be defined in terms of L ...

Theorem (This paper, 2020)

There is an algorithm which takes as input a closed L-term t and which,
in time polynomial in the number of β-steps of t
and the size of t
and space linear in the size of the largest term in the reduction
outputs a heap containing a term u such that the unfolding of u is
the normal form of t.

Unfolding takes time polynomial in number of β-steps and size of result.

For decision functions, evaluation is polynomial and
P, NP, PSPACE, EXP can be defined in terms of L . . .

... but sublinear time or space is not covered

Theorem (This paper, 2020)

There is an algorithm which takes as input a closed L-term t and which, in time polynomial in the number of β-steps of t
and the size of t
and space linear in the size of the largest term in the reduction outputs a heap containing a term u such that the unfolding of u is the normal form of t.

Unfolding takes time polynomial in number of β-steps and size of result.

Size explosion

$$\text{Church numerals: } \overline{n} := \lambda fx.(\underbrace{f(f(f\cdots(f}_{n\text{ times}}\ x)\cdots))$$

$$\text{Exponentiation: } \overline{n}\ \overline{m} \succ^* \overline{m^n}$$

$$\text{Church booleans: } \overline{true} := \lambda xy.x$$

$$\text{Define: } s_E := \lambda x.\overline{true}\ \overline{true}\ (x\overline{2}(\lambda x.x))$$

$$s_E\ \bar{n} \succ^4 (\lambda y.\overline{true})\ (\underbrace{\overline{2}(\overline{2}\ldots(\overline{2}}_{n\text{ times}}(\lambda x.x)))) \underbrace{\succ \ldots \succ}_{\mathcal{O}(n)\text{ times}} (\lambda y.\overline{true})\ t_n \succ \overline{true}$$

with $\|t_n\| \in \Omega(2^n)$.

Easy theorem I

Theorem

Turing machines can simulate L with a constant-factor overhead in space using a naive substitution-based strategy.

Easy theorem I

Theorem

Turing machines can simulate L with a constant-factor overhead in space using a naive substitution-based strategy.

For time, this strategy is exponentially wrong.

Easy theorem II

Theorem

Turing machines can simulate L with a polynomial overhead in time using a heap-based strategy.

Easy theorem II

Theorem

Turing machines can simulate L with a polynomial overhead in time using a heap-based strategy.

For space, this strategy has a logarithmic factor overhead
because pointers can become too large

Pointer explosion

$$\mathsf{N} := (\lambda xy.xx)\overline{true}.$$

$$\begin{aligned} s_P := \quad & \underbrace{\mathsf{N}(\cdots(\mathsf{N}}_{n \text{ times}}\ \overline{true})\ldots) \\ \succ^n\ & \underbrace{(\lambda y.\overline{true}\,\overline{true})(\cdots((\lambda y.\overline{true}\,\overline{true})}_{n \text{ times}}\ \overline{true})\ldots) \\ \succ^{2n}\ & \overline{true} \end{aligned}$$

Pointer explosion

$$N := (\lambda xy.xx)\overline{true}.$$

$$\begin{aligned} s_P := \quad & \underbrace{N(\cdots(N}_{n \text{ times}}\ \overline{true})\ldots) \\ & \succ^n \underbrace{(\lambda y.\overline{true}\,\overline{true})(\cdots((\lambda y.\overline{true}\,\overline{true})}_{n \text{ times}}\ \overline{true})\ldots) \\ & \succ^{2n} \overline{true} \end{aligned}$$

$3n$ beta reductions $\leadsto$ $3n$ entries on the heap

Pointer explosion

$$\mathsf{N} := (\lambda xy.xx)\overline{true}.$$

$$\begin{aligned} s_P := \quad & \underbrace{\mathsf{N}(\cdots(\mathsf{N}}_{n \text{ times}}\ \overline{true})\dots) \\ & \succ^{n} \underbrace{(\lambda y.\overline{true}\,\overline{true})(\cdots((\lambda y.\overline{true}\,\overline{true})}_{n \text{ times}}\ \overline{true})\dots) \\ & \succ^{2n} \overline{true} \end{aligned}$$

$3n$ beta reductions $\rightsquigarrow$ $3n$ entries on the heap

Heap pointers are of size $\log n$, space consumption is $\Omega(n \log n)$

Substitution-based strategy is ok for space, wrong for time on size-exploding terms

Heap-based strategy is ok for time, wrong for space due to pointer explosion

Substitution-based strategy is ok for space, wrong for time
on size-exploding terms

Heap-based strategy is ok for time, wrong for space
due to pointer explosion

Size-exploding terms do not exhibit pointer explosion!

Substitution-based strategy is ok for space, wrong for time on size-exploding terms

Heap-based strategy is ok for time, wrong for space due to pointer explosion

Size-exploding terms do not exhibit pointer explosion!

(There's always enough space for the pointers)

Substitution-based strategy is ok for space, wrong for time
on size-exploding terms

Heap-based strategy is ok for time, wrong for space
due to pointer explosion

Solution:
Interleave both strategies.
For each k, run substitution-based strategy for k steps,
if size explodes run the heap-based strategy instead

In the paper

- Substitution-based stack machine verified in Coq w.r.t. time and space
- Heap-based stack machine verified in Coq w.r.t. time and space
- Sketch of a Turing machine simulating heap-based stack machine
- Sketch of a Turing machine simulating substitution-based stack machine while checking space consumption
- Detailed proof of interleaving strategy

Conclusion

The natural measures for the weak call-by-value λ-calculus

- are compositional
- for a compositional model of computation
- cover both time and space
- can be used to define standard complexity classes

Conclusion

The natural measures for the weak call-by-value λ-calculus

- are compositional
- for a compositional model of computation
- cover both time and space
- can be used to define standard complexity classes
- are feasible to use in mechanisations

Future Work

- mechanise basic complexity theory in Coq based on extraction

Future Work

- mechanise basic complexity theory in Coq based on extraction

- extension to the full λ-calculus
- extension to sublinear time and space
- prove P $\subseteq$ PSPACE without reference to sequential models
- space measure without size explosion

Future Work

- mechanise basic complexity theory in Coq based on extraction

- extension to the full λ-calculus
- extension to sublinear time and space
- prove P $\subseteq$ PSPACE without reference to sequential models
- space measure without size explosion

Questions?

Careful naive substitution

There is a Turing machine M_{subst} that, given

- a term s
- a binary number k indicating the number of β-steps
- a binary number m indicating the maximum space to use

halts in time $\mathcal{O}(k \cdot \text{poly}(\min(m, \|s\|_\mathsf{S})))$
and space $\mathcal{O}(\min(m, \|s\|_\mathsf{S}) + \log m + \log k)$
s.t. one of the following holds:

- The machine outputs a term t, then s has normal form t and $m \geq \|s\|_\mathsf{S}$ and $k \geq \|s\|_\mathsf{T}$.
- The machine halts in a state named *space bound reached* and $m \leq \|s\|_\mathsf{S}$.
- The machine halts in a state named *space bound not reached* and $k < \|s\|_\mathsf{T}$.

Hybrid strategy

1. Initialise $k := 0$ (in binary)
2. Compute $m := \|s\| \cdot p(k)$
3. Run M_{subst} on s for k β-steps with space bound m:
    - If M_{subst} computes a normal form, output it and halt.
    - If M_{subst} halts in *space bound not reached*, set $k := k + 1$ and go to (2).
    - If M_{subst} halts in *space bound reached*, continue at (4).
4. Run M_{heap} on s for k β-steps.
    - If this computes a normal form, output it and halt.
    - Otherwise, set $k := k + 1$ and go to (2).

Hybrid strategy

1. Initialise $k := 0$ (in binary)
2. Compute $m := \|s\| \cdot p(k)$
3. Run M_{subst} on s for k β-steps with space bound m:
   - If M_{subst} computes a normal form, output it and halt.
   - If M_{subst} halts in *space bound not reached*, set $k := k + 1$ and go to (2).
   - If M_{subst} halts in *space bound reached*, continue at (4).
4. Run M_{heap} on s for k β-steps.
   - If this computes a normal form, output it and halt.
   - Otherwise, set $k := k + 1$ and go to (2).

Lemma

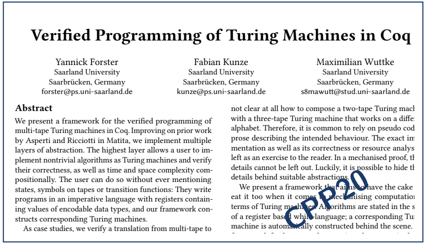
$\log \|s\|_T \in \mathcal{O}(\|s\|_S)$ – *there's always enough space to count steps in binary*

Turing machines as mechanised model of computation

Turing machines are

- the de-facto standard model for computational complexity theory
- simple to define
- easy to understand

Turing machines as mechanised model of computation


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

Abstract

We present a framework for the verified programming of multi-tape Turing machines in Coq. Improving on prior work by Asperti and Ricciotti in Matita, we implement multiple layers of abstraction. The highest layer allows a user to implement nontrivial algorithms as Turing machines and verify their correctness, as well as time and space complexity compositionally. The user can do so without ever mentioning states, symbols on tapes or transition functions: They write programs in an imperative language with registers containing values of encodable data types, and our framework constructs corresponding Turing machines.

As case studies, we verify a translation from multi-tape to

not clear at all how to compose a two-tape Turing macl with a three-tape Turing machine that works on a diffe alphabet. Therefore, it is common to rely on pseudo cod prose describing the intended behaviour. The exact im mentation as well as its correctness or resource analys left as an exercise to the reader. In a mechanised proof, tl details cannot be left out. Luckily, it is possible to hide tl details behind suitable abstractions.

We present a framework that aims to have the cake eat it too when it comes to mechanising computation terms of Turing machines. Algorithms are stated in the s of a register based while language; a corresponding Tu machine is automatically constructed behind the scene.

CPP20

Turing machines are

- the de-facto standard model for computational complexity theory
- simple to define
- easy to understand

“Turing machines as model of computation are *inherently infeasible* for the formalisation of any computability or complexity theoretic result.”