The strong invariance thesis for a λ-calculus

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

Fabian Kunze
Saarland University
Max Planck Institute for Informatics
fkunze@mpi-inf.mpg.de

Marc Roth
Saarland University
Cluster of Excellence (MMCI)
mroth@mmci.uni-saarland.de

Abstract

The model of computation most commonly used in the areas of complexity and algorithm theory is Turing machines. Time and space measure for algorithms are thus defined via the running time and space consumption of a Turing machine. However, Turing machines are notoriously hard and difficult to reason about, and the definitions are rarely used in a rigorous way.

The (strong) invariance thesis states that all reasonable models of computation also agree, to some extent, in their notions of complexity: 'Reasonable' machines can simulate each other within a polynomially bounded overhead in time and a constant-factor overhead in space."

We prove the invariance thesis between a call-by-value λ-calculus and Turing machines. We are currently formalising the proof in the proof assistant Coq.

We propose a talk on ongoing work concerning the simulation of a λ-calculus with Turing machines, establishing the invariance thesis [7] for our kind of λ-calculus. The weak invariance thesis, only concering the time of the simulation, has been established for a weak λ-calculus by Dal Lago and Martini [3] and the full λ-calculus by Accattoli and Dal Lago[1]. We base our studies on a variant of the weak λ-calculus [3]. We describe two interpretation strategies that are both on their own not sufficient to prove the strong invariance thesis, before combining them into a sufficient strategy.

While we currently formalise our work, for this talk we will focus on an informal presentation of the interpretation strategies, introducing the, to the best of our knowledge, first proof of the strong invariance thesis for any kind of λ-calculus. We hope that this enables us to formalise results in the area of complexity theory, relying on both the simple measures and powerful verification techniques available for our calculus.

1 Weak call-by-value λ-calculus L

We use a variant of Dal Lago and Martini's calculus [3] we call L, where we only allow abstraction as values:

$$\frac{}{(\lambda x.s)(\lambda y.t) \succ s[x := \lambda y.t]} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

In L, the only β-redexes are applications of abstractions and no redexes occur under binders. Reduction for L is uniformly confluent, making every reduction path to a normal form have the same length. The calculus was formalised as a computational model in order to analyse computability theory [5].

We define the time-consumption of a closed L-term as the number of steps it takes to evaluate to a normal form. I.e. if

$$s = s_0 \succ s_1 \succ \dots s_k = v$$

for a term s and an abstraction v, then

$$\mathsf{Time}(s) := k.$$

Note that this definition is well-formed because the number of steps to a normal forms does not depend on the concrete reduction path.

This differs from an approach where the size of β-redexes is incorporated into the cost measure [3]. By our definition we gain a simpler measure, enabling future formalisations of complexity theory.

We define the space consumption of a closed L-term as the size of the biggest intermediate term occuring in any reduction to the normal form:

$$\mathrm{Space}(s) := \max_{\{s_i \mid s \succ^* s_i\}} |s_i|$$

with $|x| =$ *de Bruijn index of* x, $|st| = 1 + |s| + |t|$, $|\lambda x.s| = 1 + |s|$. Note that this does not work for sub-linear computations, so all further results have to be read as excluding sub-linearity. Our space measure may result in a larger space consumption than needed in realistic implementations that use structure sharing. This allows for the in-principle very naive simulation described in Section 3.

2 Simulating Turing machines

The simulation of Turing machines in L is the easier part. Similar to Dal Lago and Martini, we use Scott-encodings and a fixed-point combinator to program in L.

Formally, we define the type of Turing machines with n additional tapes over alphabet Σ as a type consisting of

- a finite type of states Q
- a transition function $\delta : Q \times \Sigma^{n+1} \to Q \times \Sigma^{n+1} \times \{L, N, R\}$
- a start state $s : Q$
- a halting function $Q \to \mathbb{B}$

and follow Asperti and Ricciotti [2] in the definition of the semantics by looping the transition function on the initial configuration (consisting of the start state and the input tape) until a halting state is reached.

We have a formalisation in Coq [8] of the functional correctness of a simulation of Turing machines in L, which is very similar to the one presented in [3].

3 Simulating L with a constant-factor overhead in space

The most straightforward way of simulating L is as one would on paper. To do so, we follow Dal Lago and Martini and encode the de Bruijn representation of L-terms by a prefix notation with the tokens @ (to denote applications), λ (for abstractions), $\triangleright$ and | (for numbers). We encode positions of a subterm in a term by strings over the alphabet $\{@_L, @_R, \lambda\}$. For example, the term $(\lambda xy.xy)(\lambda x.x)$ has de Bruijn representation $(\lambda 10)(\lambda 0)$, resulting in the encoding $@\lambda\lambda@ \triangleright \mid \triangleright \lambda\triangleright$. The position of 1 in this term is $@_L\lambda\lambda@_L$ (i.e. it is on the left side of an application, below two lambdas, on the left side of the application below the lambdas).

To evaluate a single (leftmost) reduction step $s \succ s'$, the interpreter uses 5 additional tapes (pre, funct, arg, post, position) and proceeds as follows

$$\underbrace{\cdots}_{\text{pre}}\ @\lambda\ \underbrace{\cdots}_{\text{funct}}\ \underbrace{\lambda\cdots}_{\text{arg}}\ \underbrace{\cdots}_{\text{post}}$$

1. Find the first β-redex,
   - copy to pre until @λ is read

- copy next *complete* term to funct (and remember its position on the position tape)
- if the next token is λ, copy the next term to arg and remaining tokens to post
- otherwise, move funct onto pre and start from beginning

2. copy funct to pre, replacing bound variables with arg
3. copy post to pre

Afterwards, the content of pre contains s' and can be copied to the main working tape, and the process iterated until the first symbol is a λ.

The commonly known problem when defining the time measure for terms of λ-calculi is that a term can have multiple occurences of a variable, making the substitution-based simulation of k steps potentially exponential in k. The church encoding of 2, namely $\bar{2} = \lambda xy.x(xy)$, can double the size of a term in one step:

$$\bar{2}t \succ \lambda y.t(ty)$$

So, with $I := \lambda x.x$ the term

$$\underbrace{\bar{2}(\bar{2}(\cdots(\bar{2}\, I)\ldots))}_{k \text{ times}}$$

normalises in k steps, but takes 2^k time when interpreting with this naive substitution-based approach.

4 Simulating L with a polynomially bounded overhead in time

The issue described in the last section also occurs in the implementation of real programming languages and is circumvented by implementing closures, replacing the explicit copying of the argument by a pointer to the argument-term in some kind of storage structure (e.g. a heap). We have a formalisation of the computational equivalence proof for L and a very similar λ-calculus with explicit closures.

The main idea is that a term consists of a representation part and a heap. The representation part can contain pointers to the heap at the position of variables. For a β-reduction, the argument now has to be copied to the heap and all variables replaced by a pointer.

The pointers to a heap with k elements need size $O(k)$, which means that the iterated process of replacing the variable with a pointer is possible in polynomial time.

However, this approach has a different problem: the size of pointers may become too big. To make this clear, let $\mathrm{N} := (\lambda xy.xx)I$. Then

$$\underbrace{\mathrm{N}(\cdots(\mathrm{N}\, I)\ldots)}_{k \text{ times}} \succ^k \underbrace{(\lambda y.II)(\cdots((\lambda y.II)\, I)\ldots)}_{k \text{ times}} \succ^{2k} I$$

needs $3k$ entries on the heap (some of them containing heap addresses), resulting in quadratic space consumption. While assuming $O(k)$ space for a pointer is overestimated, storing pointers in binary will still result in non-linear ($O(k \log k)$) space consumption.

5 Simulating L

The situation for the two described approaches is that for k leftmost reductions of the form $s = s_0 \succ s_1 \succ \cdots \succ s_k$ we have:

	substitution-based	heap-based
time	$O(\sum_i \lvert s_i\rvert^2)$	$O(\mathrm{poly}(k, \lvert s\rvert, \lvert s_k\rvert))$
space	$O(\mathrm{Space}(s))$	$O(\mathrm{Space}(s) + k^2 \cdot (\lvert s\rvert + k))$

Note that because $|s_i|$ can be exponential in k, the time of the substitution-based interpreter is exponential, but the space consumption has only constant-factor overhead. The $|s_k|$ in the running time of the heap-based interpreter is needed to unfold the closure into a plain L term encoding. As encoded booleans have constant size, this is no problem for decision functions used in complexity theory.

For some families of terms where the size depends on k, the factor $|s|+k$ in the space consumption of the heap-based interpreter leads to a non-linear space overhead, while the time overhead is polynomially bounded.

This is sufficient to simulate L with polynomially bounded overhead in time and to simulate L with constant-factor overhead in space, but not sufficient to simulate with both constraints fulfilled at the same time.

However, the problematic cases turn out to be mutually exclusive. We can thus build an interpreter fulfilling the invariance thesis by dovetailing the two previous interpreters.

As the number of β-reductions from a term s to a normal form is not known initially, we execute the following strategy while iteratively incrementing k: The substitution-based interpreter is executed until the space consumption exceeds $k^2 \cdot (|s| + k)$ or a normal form is found. This space constraint can be checked on the fly in the given bounds. Afterwards, the heap-based approach is used on s for exactly k steps. The process is repeated with increased k if this did not yield a normal form.

This results in a Turing-machine interpreter for L with a polynomially bounded overhead in time and a constant-factor overhead in space, establishing the strong invariance thesis between L and Turing machines.

6 Formalisation

We already formalised the functional correctness of both simulation approaches, however not as Turing machines, but as simple end-recursive Coq-functions using lists as stacks. We also have a functionally correct Turing machine interpreter in L, extracted from the functions in the definition of Turing machines by a generic Coq-to-L extraction framework [4]. We are currently enriching the framework by semi-automatic time-complexity. The following table gives an overview over the proof size:

	spec	proof
Functional correctness of L-interpreters	1192	1390
L-extraction framework	1316	610
TM-interpreter (no verified complexity analysis)	388	335

The largest remaining part is to verify Turing machines executing the verified algorithms. We are currently investigating whether a relational approach [2] can be mechanised with methods used for while-programs by Pous [6] or whether a different approach using for instance separation logic is more promising.

References

[1] Beniamino Accattoli and Ugo Dal Lago. (Leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Computer Science*, 12(1), 2016.

[2] Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, October 2015.

[3] Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.

[4] Yannick Forster and Fabian Kunze. Verified extraction from Coq to a lambda-calculus. *Coq workshop 2016, https://www.ps.uni-saarland.de/~forster/coq-workshop-16/*, 2016.

[5] Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. *unpublished, submitted for review, https://www.ps.uni-saarland.de/extras/L-computability/*, 2017.

[6] Damien Pous. Kleene algebra with tests and coq tools for while programs. In *ITP 2013*, pages 180–196, 2013.

[7] Cees Slot and Peter van Emde Boas. The problem of space invariance for sequential machines. *Information and Computation*, 77(2):93 – 122, 1988.

[8] The Coq Proof Assistant. http://coq.inria.fr.