The Invariance Thesis for a $\lambda$-Calculus
Towards Formal Complexity Theory

Yannick Forster  Fabian Kunze  Marc Roth

Research Immersion Lab Talk
Advisor: Prof. Dr. Gert Smolka

13.01.2017
Related Work

Formal correctness proofs for TMs are tedious:

- not compositional
- few abstractions (data encoding with finite alphabet, no local variables,...)

Andrea Asperti and Wilmer Ricciotti
A formalization of multi-tape Turing machines
Theoretical Computer Science, 2015

Eelis van der Weegen, James McKinna
A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq
TYPES 2008

Ugo Dal Lago and Simone Martini
The Weak Lambda Calculus as a Reasonable Machine
Theoretical Computer Science, 2008
L: Weak Call-by-Value $\lambda$-Calculus

L: Syntax and Semantics

$s, t ::= x \mid \lambda s \mid s \cdot t \quad (x \in \mathbb{N})$

\[
\begin{align*}
    s & \succ s' \\
    s \cdot t & \succ s' \cdot t \\
    t & \succ t' \\
    s \cdot t & \succ s \cdot t' \\
    (\lambda s)(\lambda t) & \succ s_{\lambda t}^0
\end{align*}
\]

- data represented by abstractions (Scott encoding)
- recursion using fixpoint combinator

$\Rightarrow$ Turing complete
The Invariance Thesis

Definition (Invariance Thesis)

"Reasonable" machines can simulate each other within a polynomial bounded overhead in time and a constant-factor overhead in space.

Ensures consistency w.r.t classes closed under poly-time/constant-space reductions.
The Complexity Measures

Consider a term $s$ with the evaluation

$$s = s_0 \succ s_1 \succ \cdots \succ s_k$$

The *time consumption* of $s$ is the number of reduction steps:

$$\text{Time}_s := k$$

The *space consumption* of $s$ is the maximum of the sizes of intermediate terms of all possible evaluations:

$$\text{Space}_s := \max_{i=0,\ldots,k} |s_i|$$
Encoding Terms

- terms: prefix notation
- Positions: strings over \{@_L, @_R, \lambda\}

**Example**

\((\lambda xy.x y)(\lambda x.x) \approx (\lambda \lambda 10) (\lambda 0)\) is encoded by string \(_@\lambda \lambda @_\triangleright \triangleright \lambda \triangleright\). In this term, '1' occurs at position \(_@L \lambda \lambda @_L\)
The Naive Interpreter

Idea: evaluate as one would on paper
For one step $s \Rightarrow s'$:

1. find the first $\beta$-Redex and split $s$ onto 4 tapes:

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>@λ</td>
<td>funct</td>
<td>arg</td>
</tr>
</tbody>
</table>
```

- copy to pre until $@\lambda$ is read
- copy next *complete* term to funct (with additional position tape)
- if next token is $\lambda$, copy 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, reduced term $s'$ is in pre

Per step: $O(|s| + |s'|)$ time & $O(|s| + |s'|)$ space
For whole evaluation: $O(\sum_i |s_i|)$ time & $O(max_i |s_i|)$ space
Exponentially Large Terms

\( \bar{2} := \lambda xy.x (x y) \) can double the size of a term in one step:

\[
\bar{2} t \Rightarrow \lambda y.t (t y)
\]

So, with \( I := \lambda x.x \):

\[
\bar{2} (\bar{2} (\cdots (\bar{2} I) \cdots)) \\
\text{k times}
\]

normalizes in \( k \) L-steps, but takes \( \Omega(2^k) \) time for the naive interpreter

⇒ other interpreter needed.
The heap-based Interpreter

Use environments on a heap to delay substitutions:

- call (thunk) \( c = s\langle E \rangle \): pair of encoded L-term \( s \) and heap-address \( E \)
- heap \( H \): list of entries (\( \bot \) or \( c \# E' \)), addressed by position.
- call stack \( CS \): list of tuples (\( @_L, c \)) or (\( @_R, c \)) (for \( @_R \), \( c \) fully reduced)
- interpreter state: current call \( CC \), \( CS \) and \( H \).
- initial state: \( CC = s\langle 0 \rangle \), \( CS = [\] \) and \( H = [\bot] \)

Example

The result of \((\lambda x. x) ((\lambda xy. x y) (\lambda x. x)) \triangleright (\lambda x. x) (\lambda x. x y)_{\lambda x.x}^y\) is represented by

\[
CC = (\lambda @ \triangleright |\triangleright\rangle \langle 1
CS = [(@_R, (\lambda \triangleright \rangle \langle 0\rangle)]
H = [\bot, (\lambda \triangleright \rangle \langle 0\rangle) \# 0]
\]
The heap-based Interpreter (2)

Each step of the interpreter depends on the current call $CC = s \langle E \rangle$:

- if $s = s_L s_R$: push $(@_L, s_R[E])$ on $CS$ and set $CC$ to $s_R \langle E \rangle$
- if $s = x$: get new $CC$ by lookup of $x$ in $E$
- if $s = \lambda s'$:
  - if $CS$ is empty: the term is fully evaluated
  - if $CS = (@_L, c_R) :: CS'$: set $CC := c_R$ and put $(@_R, CC)$ on stack instead.
  - if $CS = (@_R, \lambda t \langle E' \rangle) :: CS'$: store $s_R \langle E \rangle \# E'$ on heap as $\hat{E}$ and set $CC := t \langle \hat{E} \rangle$

Observations for evaluation $s_0 \succ s_1 \succ \cdots \succ s_k$:

- all calls contain subterms of $s$
- Heap contains $\# H = k + 1$ elements, each of size $\leq |s| + 2 \cdot \log(\# H)$
- $CS$ & $CC$ representing $s_i$ have size $O(|s_i|)$

$\Rightarrow$ space consumption: $O((\max_i |s_i|) + k \cdot (|s| + \log(k)))$

- time per interpreter step: $O(|s_i| \cdot \# H + CC + CS)$
- amortized, $\text{poly}(|s_0|)$ interpreter-steps per $\beta$-reduction.

$\Rightarrow$ time consumption: $O(\text{poly}(k, |s_0|))$
Sub-linear-logarithmic Small Terms

Let $N := (\lambda xy.x x) I$, then

$$N (\cdots (N I) \ldots)$$

let $k$ times

$$\prec^k ((\lambda y. I I) (\cdots ((\lambda y. I I) I) \ldots)$$

let $k$ times

$$\prec 2^k I$$

 Needs $3k$ entries (with addresses) on heap, but definition permits only $O(k)$ space
Consider evaluation $s = s_0 \succ s_1 \succ \cdots \succ s_k$:

<table>
<thead>
<tr>
<th></th>
<th>naive</th>
<th>heap-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time$_s = k$</td>
<td>$O(\sum_i</td>
<td>s_i</td>
</tr>
<tr>
<td>Space$_s = \max_i</td>
<td>s_i</td>
<td>$</td>
</tr>
</tbody>
</table>

- If $\text{Space}_s \geq k^2 \cdot (|s| + \log(k))$, use heap-based interpreter.
- Otherwise, use naive interpreter.
- archived by increasing bound on space of naive interpreter

$\Rightarrow$ the simulation respects the Invariance Thesis (assuming $k > |s|$)
restricted Gallina:

- functions: operating on tuples/records, representing tapes
- data: (list of) tokens and natural numbers
- recursion: tail-recursive or explicit iteration of step-function
Formalization (2)

<table>
<thead>
<tr>
<th></th>
<th>spec</th>
<th>proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-interpreters</td>
<td>1192</td>
<td>1390</td>
</tr>
<tr>
<td>L-extraction framework</td>
<td>1316</td>
<td>610</td>
</tr>
<tr>
<td>TM-interpreter</td>
<td>388</td>
<td>335</td>
</tr>
<tr>
<td>TMs</td>
<td>2254</td>
<td>2336</td>
</tr>
</tbody>
</table>

just functional specification
partly supporting time-analysis
no complexity analysis

Fabian Kunze
The Invariance Thesis for a λ-Calculus
13.01.2017
Formalizing the Naive Interpreter

**Lemma** enc_decompose \((s: \text{list} \ \text{token}) \ (pos: \text{option} \ (\text{list} \ \text{posToken}))\):
\[
\text{validOptPos} \ s \ pos \rightarrow \text{enc} \ s = \text{leftOf} \ s \ pos \ ++ \ \text{encAt} \ s \ pos \ ++ \ \text{rightOf} \ s \ pos.
\]

**Inductive** nextPos : \(\text{term} \rightarrow \text{position} \rightarrow \text{option} \ \text{position} \rightarrow \text{Prop} :=
\]
\[
| \ \text{nextPos} \text{AppInit} \ s \ t: \text{nextPos} \ (\text{app} \ s \ t) \ [] \ (\text{Some} \ [\text{posAppL}])
(*2 \ more \*)
\]
\[
| \ \text{nextPos} \text{LamSome} \ s \ p1 \ p2: \text{nextPos} \ s \ p1 \ (\text{Some} \ p2)
\rightarrow \text{nextPos} \ (\text{lam} \ s) \ (\text{posLam::p1}) \ (\text{Some} \ (\text{posLam::p2}))
(*5 \ more \ congruences*)
\]
\[
| \ \text{nextPos} \text{closeScope} \ s \ t \ p1: \text{nextPos} \ s \ p1 \ \text{None}
\rightarrow \text{nextPos} \ (\text{app} \ s \ t) \ (\text{posAppL::p1}) \ (\text{Some} \ [\text{posAppR}]).
\]

**Lemma** nextPos_leftOf’ \(s \ \text{pos} \ p’ \ a \ \text{rem}:
nextPos \ s \ \text{pos} \ p’ \rightarrow \text{enc} \ (\text{getAt} \ s \ \text{pos}) = a::\text{rem}
\rightarrow \text{leftOf’} \ s \ \text{pos++}[a] = \text{leftOf} \ s \ p’.
Fixpoint nextTerm’ res rem (stack: option position) :=
    match stack, rem with
    | None, _ ⇒ (res,rem,stack)
    | Some stack’, a::rem’ ⇒ nextTerm’ (res++[a]) rem’ (updateStack stack’ a)
end.

Lemma nextTerm’_correct res rem pos s stack’:
    validPos s pos
    → nextTerm’ res (enc (getAt s pos)++rem) (Some (rev pos ++ stack’))
    = nextTerm’ (res++enc (getAt s pos)) rem (closeScopeStack (rev pos ++ stack’)).

Definition nextTerm rem := nextTerm’ [] rem (Some []).

Lemma nextTerm_correct s rem : nextTerm (enc s++rem) = (enc s,rem,None).
Definition searchRedex_step (comp : searchRedex_state) : searchRedex_state := (*...*).

Inductive searchRedex_inv s comp : Prop :=
| notFound pos: mayFirstRedex s pos → current comp = remTerm s pos → preredex comp = leftOf s pos (*...*) → searchRedex_inv s comp
| foundRedex pos : firstRedex s = Some pos → preredex comp = leftOf s (Some pos) → functional comp = enc(getAt s (pos++[posAppL;posLam])) (*...*) → searchRedex_inv s comp.

Lemma searchRedex_step_correct s comp:
(* invariant preserved *) ∧ (* current comp decreases *).

Definition searchRedex (comp:searchRedex_state) :=
loop (|current comp|) searchRedex_step (fun comp’ ⇒ Dec (current comp’ = [])) comp.

Lemma searchRedex_correct’ s comp :
searchRedex_inv s comp
→ ∃ comp’, searchRedex comp = Some comp’ ∧ current comp’ = [] ∧ searchRedex_inv s comp’.
Summary

The cbv $\lambda$-Calculus is as reasonable for complexity theory as Turing machines

Possible future work:

- Formalize the complexity analysis
- Complexity theory using L: NP, many-one-reductions, hierarchy theorems...

Thanks!