The Invariance Thesis for a λ-Calculus

Towards Formal Complexity Theory

Yannick Forster   Fabian Kunze   Marc Roth

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

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Programming Systems Lab

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 λ-Calculus

L: Syntax and Semantics

$$s, t \ ::= \ x \ | \ \lambda s \ | \ s\,t \qquad (x \in \mathbb{N})$$

$$\frac{s \succ s'}{s\,t \succ s'\,t} \qquad \frac{t \succ t'}{s\,t \succ s\,t'} \qquad \frac{}{(\lambda s)\,(\lambda t) \succ s^0_{\lambda t}}$$

- 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:

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

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

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

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 $@\lambda\lambda@ \triangleright \mid \triangleright \lambda\triangleright$.
In this term, '1' occurs at position $@_L\lambda\lambda@_L$

The Naive Interpreter

Idea: evaluate as one would on paper

For one step $s \succ s'$:

1. find the first β-Redex and split s onto 4 tapes: $\underbrace{\cdots}_{\text{pre}} @\lambda \underbrace{\cdots}_{\text{funct}} \underbrace{\lambda \cdots}_{\text{arg}} \underbrace{\cdots}_{\text{post}}$
    - copy to pre until $@\lambda$ is read
    - copy next *complete* term to funct (with additional position tape)
    - if next token is λ, 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: $\mathcal{O}(|s| + |s'|)$ time & $\mathcal{O}(|s| + |s'|)$ space

For whole evaluation: $\mathcal{O}(\sum_i |s_i|)$ time & $\mathcal{O}(max_i\, |s_i|)$ space

Exponentially Large Terms

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

$$\overline{2}\,t \succ \lambda y.t\,(t\,y)$$

So, with $\mathbf{I} := \lambda x.x$:

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

normalizes in k L-steps, but takes $\Omega(2^k)$ time for the naive interpreter
$\Rightarrow$ 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 $(@_\mathrm{L}, c)$ or $(@_\mathrm{R}, c)$ (for $@_\mathrm{R}$, 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 $(\lambda x.x)\,((\lambda xy.x\,y)\,(\lambda x.x)) \succ (\lambda x.x)\,\underline{(\lambda x.x\,y)^{y}_{\lambda x.x}}$ is represented by

$$\begin{aligned} CC &= (\lambda @ \triangleright |\triangleright)\langle 1\rangle \\ CS &= [(@_\mathrm{R}, (\lambda\triangleright)\langle 0\rangle)] \\ H &= [\bot, (\lambda\triangleright)\langle 0\rangle)\#0] \end{aligned}$$

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 $(@_\mathrm{L}, s_R[E])$ 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 $CS = (@_\mathrm{L}, c_R) :: CS'$: set $CC := c_R$ and put $(@_\mathrm{R}, CC)$ on stack instead.
  - if $CS = (@_\mathrm{R}, \lambda t\langle E'\rangle) :: CS'$: 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 $\#\mathrm{H} = k+1$ elements, each of size $\leq |s| + 2\cdot\log(\#\mathrm{H})$
- CS & CC representing s_i have size $\mathcal{O}(|s_i|)$

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

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

$\Rightarrow$ time consumption: $\mathcal{O}(\mathrm{poly}(k, |s_0|))$

Sub-linear-logarithmic Small Terms

Let $\mathsf{N} := (\lambda xy.x\,x)\,\mathbf{I}$, then

$$
\begin{aligned}
&\underbrace{\mathsf{N}\,(\cdots(\mathsf{N}}_{k\text{ times}}\,\mathbf{I})\ldots)\\
&\succ^{k} \underbrace{(\lambda y.\mathbf{I}\,\mathbf{I})\,(\cdots((\lambda y.\mathbf{I}\,\mathbf{I})}_{k\text{ times}}\,\mathbf{I})\ldots)\\
&\succ^{2k}\,\mathbf{I}
\end{aligned}
$$

Needs $3k$ entries (with addresses) on heap, but definition permits only $\mathcal{O}(k)$ space

Complexity Overview

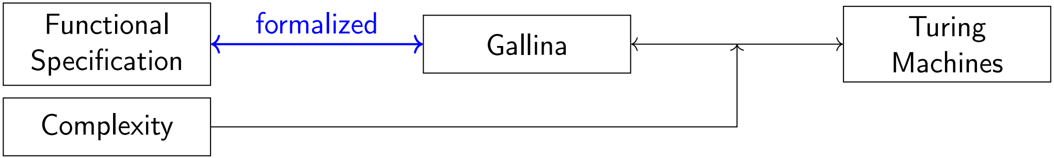
Consider evaluation $s = s_0 \succ s_1 \succ \cdots \succ s_k$:

	naive	heap-based
$\text{Time}_s = k$	$\mathcal{O}(\sum_i \lvert s_i \rvert^2)$	$\mathcal{O}(\text{poly}(k, \lvert s \rvert))$
$\text{Space}_s = \max_i \lvert s_i \rvert$	$\mathcal{O}(\text{Space}_s)$	$\mathcal{O}(\text{Space}_s + k^2 \cdot (\lvert s \rvert + \log(k)))$

- 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 $k > |s|$)

Formalization



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)

	spec	proof	
L-interpreters	1192	1390	just functional specification
L-extraction framework	1316	610	partly supporting time-analysis
TM-interpreter	388	335	no complexity analysis
TMs	2254	2336	

Formalizing the Naive Interpreter

```
Lemma enc_decompose (s: list token) (pos: option (list posToken)):
  validOptPos s pos
    → enc s = leftOf s pos ++ encAt s pos ++ rightOf s pos.

Inductive nextPos : term → position → option position → Prop :=
| nextPos_AppInit s t: nextPos (app s t) [] (Some [posAppL])
  (*2 more *)
| nextPos_LamSome s p1 p2: nextPos s p1 (Some p2)
    → nextPos (lam s) (posLam::p1) (Some (posLam::p2))
  (*5 more congruences*)
| nextPos_closeScope s t p1: nextPos s p1 None
    → nextPos (app s t) (posAppL::p1) (Some [posAppR]).

Lemma nextPos_leftOf' s pos p' a rem:
  nextPos s pos p' → enc (getAt s pos) = a::rem
    → leftOf' s pos++[a] = leftOf s p'.
```

Copying whole Subterms

```
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).
```

Finding Redexes

```
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 λ-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!