A Graph-Rewriting Perspective of the Beta-Law

Dan R. Ghica
Todd Waugh Ambridge
(University of Birmingham)

Koko Muroya
(University of Birmingham & RIMS, Kyoto University)
Call-by-value beta-law

\[(\lambda x. t) \ v = t [v/x]\]

Terms \( t ::= x | \lambda x. t | tt | \ldots \)

Values \( v ::= x | \lambda x. t | \ldots \)

*golden standard* of (functional) program equivalence and compiler optimisation

“A function can be applied to a value before evaluation without changing the outcome”
Call-by-value beta-law

\[(\lambda x.t) \nu = t[\nu/x]\]

terms \( t ::= x | \lambda x.t | tt | \ldots \)

values \( \nu ::= x | \lambda x.t | \ldots \)

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions
Call-by-value beta-law

\[(\lambda x.t) \nu = t[\nu/x] \]

terms \( t ::= x | \lambda x.t | tt | n | \text{succ}(n) | ... \)

values \( \nu ::= x | \lambda x.t | n | ... \)

**golden standard** of (functional) program equivalence and compiler optimisation

... respected by most **intrinsic/extrinsic** language extensions
Call-by-value beta-law

\[(\lambda x. t) \nu = t [\nu/x]\]

Terms \( t \) := \( x \mid \lambda x. t \mid tt \mid \langle t, t \rangle \mid \text{fst}(t) \mid \text{snd}(t) \mid \ldots \)

Values \( \nu \) := \( x \mid \lambda x. t \mid \langle \nu, \nu \rangle \mid \ldots \)

*golden standard* of (functional) program equivalence and compiler optimisation

… respected by most *intrinsic/extrinsic* language extensions

*algebraic data structures*
Call-by-value beta-law

\[(\lambda x. t) \nu = t [\nu/x]\]

terms \( t ::= x \mid \lambda x. t \mid tt \mid \mu x. t \)

values \( \nu ::= x \mid \lambda x. t \mid \ldots \)

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions
Call-by-value beta-law

\[ (\lambda x.t)\, v = t\, [v/x] \]

terms \( t ::= x \mid \lambda x.t \mid tt \mid \text{if } t \text{ then } t \text{ else } t \)

values \( v ::= x \mid \lambda x.t \mid \ldots \)

golden standard of (functional) program equivalence and compiler optimisation

… respected by most intrinsic/extrinsic language extensions
Call-by-value beta-law

\[(\lambda x. t) \nu = t \subst{\nu}{x}\]

terms \( t ::= x | \lambda x. t | tt | \text{op}(t, \ldots, t) | \ldots \)

values \( \nu ::= x | \lambda x. t | \ldots \)

golden standard of (functional) program equivalence and compiler optimisation

\ldots respected by most intrinsic/extrinsic language extensions

algebraic effects & handlers
Call-by-value beta-law

$$(\lambda x. t) \nu = t[\nu/x]$$

terms $t ::= x | \lambda x. t | tt | \text{call}(cc(t)) | \ldots$

values $\nu ::= x | \lambda x. t | \ldots$

*golden standard* of (functional) program equivalence and compiler optimisation

... respected by most *intrinsic/extrinsic* language extensions
Call-by-value beta-law

\[(\lambda x.t) \nu = t [\nu/x]\]

terms \( t ::= x \mid \lambda x.t \mid tt \mid \ldots \)

values \( \nu ::= x \mid \lambda x.t \mid \ldots \)

golden standard of (functional) program equivalence and compiler optimisation

\(\ldots\) respected by most intrinsic/extrinsic language extensions

justification by (operational) semantics, but how?
Call-by-value beta-law

\[ (\lambda x . t) \, v = t \, [v/x] \]

terms \( t ::= x \mid \lambda x . t \mid tt \mid \ldots \)

values \( v ::= x \mid \lambda x . t \mid \ldots \)

*golden standard* of (functional) program equivalence and compiler optimisation

... respected by most *intrinsic/extrinsic* language extensions

justification by (operational) semantics, *but how*?
Question

**Given** an extension of untyped $\lambda$-calculus,

**what semantic property** of the extension

**validates** the call-by-value beta-law?
Question

**Given** an extension of untyped λ-calculus,

**what operational-semantic property** of the extension

**validates** the call-by-value beta-law?
Question

Given an extension of untyped λ-calculus, what *operational-semantic property* of the extension validates the call-by-value beta-law?

Answer?
Question

Given an extension of untyped $\lambda$-calculus,

what *operational-semantic property* of the extension

validates the call-by-value beta-law?

Answer?

A formal answer is yet to be stated…

But a *graph-rewriting perspective* provides:

- a useful & robust method
- key observations
Methodology

Given an operational semantics of an extended λ-calculus:

1. **Define** the contextual equivalence by:
   
   \[ t \equiv t' \iff \forall C \text{ s.t. } C[t] \text{ and } C[t'] \text{ are closed,} \]
   
   \[ C[t] \Downarrow k \iff C[t'] \Downarrow k' \]
   
   Moreover, \( k = k' \)

2. **Prove** the beta-law:
   
   \[ (\lambda x.t) \, v \equiv t[v/x] \]

3. **Observe** some sufficient condition.
Methodology

Given an operational semantics of an extended λ-calculus:

- define the contextual equivalence by:
  \[ t \triangleq t' \iff \forall C \text{ s.t. } C[t] \text{ and } C[t'] \text{ are closed, } C[t] \downarrow k \iff C[t'] \downarrow k' \]
  Moreover, \( k = k' \)

- prove the beta-law:
  \[
  (\lambda x . t) \nu \triangleq t[\nu/x]
  \]

- and observe some sufficient condition.
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step reduction
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step reduction

\[ t \downarrow k \iff t \rightarrow^* k \]

... obscures a sub-term of interest :-(

\[ C[t] \equiv E_0[n_0] \rightarrow u_1 \equiv E_1[n] \rightarrow u_2 \equiv \ldots \]

redex searching
(i.e. decomposition into evaluation context & redex)
observes `t`
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step "token-guided" graph-rewriting

closed term

basic constant
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step “token-guided” graph-rewriting

t \downarrow k \iff t^+ \rightarrow^* \text{ for some } \Delta.
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

Small-step “token-guided” graph-rewriting

closed term

distinguished edge/node as “token”

t graph representation with unique open edge
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step “token-guided” graph-rewriting

- redex searching (moving the token)
- rewriting (replacing a sub-graph)
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

Small-step "token-guided" graph-rewriting

- redex searching (moving the token)
- rewriting (replacing a sub-graph)

Garbage
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

small-step "token-guided" graph-rewriting

\[
(t \downarrow k \triangleq t^+) \rightarrow^* (k^+ \uparrow \gamma) \quad \text{for some } \gamma.
\]

... keeps a sub-term of interest traceable :-)

\[
(c \llbracket t \rrbracket)^+ = c^+ \ldots \frac{t^+}{t^+}
\]
Which operational semantics?

- easy to extend (esp. by nondeterminism, observables)
- easy to prove a contextual equivalence

Small-step "token-guided" graph-rewriting

- visible interaction between the token △ and a sub-graph
  - redex searching
  - rewriting

Step-wise reasoning to prove a contextual equivalence
Methodology

Given operational semantics of an extended λ-calculus: define the contextual equivalence by:

\[
\forall C \text{ s.t. } C[t] \text{ and } C[t'] \text{ are closed, }
\]

\[
C[t] \downarrow k \iff C[t'] \downarrow k'
\]

Moreover, \( k = k' \)

prove the beta-law:

\[
(\lambda x.t) \nu \equiv t[\nu/x]
\]

and observe some sufficient condition.
Case study: linear $\lambda$-calculus + “linear” recursion

Given operational semantics:

define the contextual equivalence by:

prove the beta-law:

and observe some sufficient condition.
Case study: linear λ-calculus + “linear” recursion

Given operational semantics:

- **closed term**
- **linear variable**
- **basic constant**
- “linear” recursion
Case study: linear λ-calculus + “linear” recursion

Given operational semantics:

Closed term

define the contextual equivalence by:

prove the beta-law:

and observe some sufficient condition.
Case study: linear λ-calculus + “linear” recursion

... prove the beta-law:

same “graph-context” with matching interface

no garbage created
Case study: linear $\lambda$-calculus + “linear” recursion

... prove the beta-law by step-wise reasoning:

- same “graph-context” with matching interface

- no garbage created
Case study: linear $\lambda$-calculus + “linear” recursion

... **prove** the beta-law *by step-wise reasoning*:

1. redex searching “within” graph-context
2. rewriting “in” graph-context
3. visiting the hole

same “graph-context” with matching interface
Case study: linear $\lambda$-calculus + “linear” recursion

1. redex searching “within” graph-context (1/6)

[Diagrams showing the process of redex searching within a graph-context with annotations indicating where the searching stopped at a value.]
Case study: linear λ-calculus + “linear” recursion

1. redex searching “within” graph-context (2/6)
Case study: linear λ-calculus + “linear” recursion

1. redex searching “within” graph-context (3/6)

![Diagram showing redex searching process](image-url)
Case study: linear λ-calculus + “linear” recursion

1. reduct searching “within” graph-context (4/6)

- Right-to-left call-by-value
Case study: linear $\lambda$-calculus + “linear” recursion

1. redex searching “within” graph-context (5/6)

![Diagram showing redex searching in a graph-context with right-to-left call-by-value.]

Muroya (U. B’ham. & RIMS, Kyoto U.)
Case study: linear $\lambda$-calculus + “linear” recursion

1. redex searching “within” graph-context (6/6)

![Diagram showing redex searching within graph-context](image)
Case study: linear $\lambda$-calculus + “linear” recursion

1. redex searching within graph-context (6 cases)

observation: only one node is inspected at each step
Case study: linear $\lambda$-calculus + “linear” recursion

2. rewriting “in” graph-context (1/3)

call-by-value
beta-reduction
Case study: linear λ-calculus + “linear” recursion

2. rewriting “in” graph-context (2/3)
Case study: linear $\lambda$-calculus + “linear” recursion

2. rewriting “in” graph-context

observation: the hole is not involved
Case study: linear λ-calculus + “linear” recursion

2. rewriting “in” graph-context (3/3)

reduction for recursion
2. rewriting “in” graph-context (3/3)

`G` contains:

- all reachable nodes from `μ`
- hence,
  - none of the hole
  - or, all of the hole

reduction for recursion
2. rewriting “in” graph-context (3/3)

the hole is

- not involved
- or, duplicated as a part of `G`

reduction for recursion
Case study: linear λ-calculus + “linear” recursion

2. rewriting “in” graph-context

**observation:** the hole is not involved, or is duplicated as a whole

**observation 2:** each rewriting step is “history-free”
Case study: linear \( \lambda \)-calculus + “linear” recursion

3. visiting the hole \((1/1)\)
Case study: linear λ-calculus + “linear” recursion

3. visiting the hole (1/1)

right-to-left call-by-value
Case study: linear $\lambda$-calculus + “linear” recursion

3. visiting the hole (1/1)

searching stopped at a value
Case study: linear $\lambda$-calculus + “linear” recursion

3. visiting the hole (1/1)

right-to-left call-by-value
Case study: linear λ-calculus + “linear” recursion

3. visiting the hole (1/1)

searching stopped at a value
Case study: linear $\lambda$-calculus + “linear” recursion

3. visiting the hole (1/1)

a redex found
Case study: linear λ-calculus + “linear” recursion

3. visiting the hole (1/1)

\( \beta \)-reduction
Case study: linear $\lambda$-calculus + “linear” recursion

3. visiting the hole (1/1)
Case study: linear λ-calculus + “linear” recursion

3. visiting the hole (1 case)

observation: the hole is reduced
Case study: linear λ-calculus + “linear” recursion

**Given** operational semantics:

Given operational semantics:

\[ t \Downarrow k \iff t^* \Downarrow \Delta \text{ for some } \Delta \]

**define** the contextual equivalence by:

\[ t \simeq t' \iff \forall C \text{ s.t. } C[t] \text{ and } C[t'] \text{ are closed,} \]

\[ C[t] \Downarrow k \iff C[t'] \Downarrow k' \]

Moreover, \( k = k' \)

**prove** the beta-law by **step-wise reasoning**:

\[ (\lambda x.t) \nu \simeq t[\nu/x] \]

and **observe** some sufficient condition.
Case study: linear λ-calculus + “linear” recursion

… prove the beta-law by step-wise reasoning,

and observe that:

1. *redex searching* only inspects one node at each step
2. *rewriting* preserves, duplicates or simply reduces a beta-redex.
3. *rewriting* is “history-free”.
Case studies so far

... prove the beta-law by step-wise reasoning,

and observe that:

1. redex searching only inspects one node at each step
2. rewriting preserves, duplicates or simply reduces a beta-redex.
3. rewriting is “history-free”.

✓ untyped pure λ-calculus
✓ basic operations, recursion, if-statement
✓ control operators: call/cc, shift/reset
• algebraic effects & handlers

method needs to be slightly adjusted
Question

Given an extension of untyped λ-calculus, what operational-semantic property of the extension validates the call-by-value beta-law?

Answer?

A formal answer is yet to be stated…

But a graph-rewriting perspective provides:

- a useful & robust method
- key observations