On the relation between call-by-value and call-by-name

Dylan McDermott

Reykjavik University

Joint work with Alan Mycroft (Cambridge)
Goal

Suppose we replace one evaluation order (e.g. call-by-value) with another (e.g. call-by-name)

How does this affect the behaviour of programs?

- No side-effects: nothing changes
- Recursion, exceptions, state, nondeterminism, . . . : ???
Call-by-value (CBV) and call-by-name (CBN)

CBV: \[ e e' \xrightarrow{\ast}^v (\lambda x. e'') e' \xrightarrow{\ast}^v (\lambda x. e'') v \xrightarrow{v} e''[x \mapsto v] \]

CBN: \[ e e' \xrightarrow{\ast}^n (\lambda x. e'') e' \xrightarrow{\ast}^n e''[x \mapsto e'] \]
Call-by-value (CBV) and call-by-name (CBN)

CBV: \[ e \ e' \xrightarrow{\ast}_v (\lambda x. e'') e' \xrightarrow{\ast}_v (\lambda x. e'') \nu \xrightarrow{\nu} e''[x \mapsto \nu] \]

CBN: \[ e \ e' \xrightarrow{\ast}_n (\lambda x. e'') e' \xrightarrow{\ast}_n e''[x \mapsto e'] \]

So if \( \Omega \) reduces to itself:

- \((\lambda x. 42) \Omega \) doesn’t terminate in CBV:
  \[ (\lambda x. 42) \Omega \xrightarrow{\nu} (\lambda x. 42) \Omega \xrightarrow{\nu} (\lambda x. 42) \Omega \xrightarrow{\nu} \ldots \]

- But does terminate in CBN:
  \[ (\lambda x. 42) \Omega \xrightarrow{n} 42 \]

CBV and CBN don’t have the same behaviour in general
CBV and CBN don’t in general have the same behaviour

But for programs\(^1\):

- No side-effects: CBV and CBN are the same
- Only recursion: if CBV terminates, then CBN terminates with same result
- Only nondeterminism: if CBV can terminate with result \(v\), CBN can terminate with result \(v\)

How can we prove these?

\(^{1}\text{Program = closed expression of ground type}\)
Method

Use an intermediate language that captures various evaluation orders:

1. Translate from source language to intermediate language

\[ e \rightarrow \langle e \rangle^v \rightarrow \langle e \rangle^n \]

source expression

\( \langle e \rangle^v \)
call-by-value intermediate term

\( \langle e \rangle^n \)
call-by-name intermediate term

2. Prove relationship between two translations

\[ \langle e \rangle^v \preceq_{\text{ctx}} \langle e \rangle^n \]
Method

Use an intermediate language that captures various evaluation orders:

1. Translate from source language to intermediate language

\[ e \rightarrow \langle e \rangle^v \rightarrow \langle e \rangle^n \]

- call-by-name intermediate term
- call-by-value intermediate term

source expression

2. Prove relationship between two translations

\[ \langle e \rangle^v \preceq_{ctx} \phi(\langle e \rangle^n) \]

Subtlety: two translations have different types

\[ \langle e \rangle^n \rightarrow \phi(\langle e \rangle^n) \]

another intermediate term
Call-by-push-value [Levy ’99]

Split syntax into **values** and **computations**

- Values don’t have side-effects, computations might
Call-by-push-value [Levy ’99]

Split syntax into values and computations

- Values don’t have side-effects, computations might

Evaluation order is explicit

- Can put two computations together: if $M_1, M_2$ are computations then

  $$M_1 \text{ to } x. M_2$$

  is a computation

- Can thunk computations: if $M$ is a computation then

  $$\text{thunk } M$$

  is a value

=> can do call-by-value and call-by-name
Call-by-push-value syntax

Value types:

\[ A, B ::= \ldots \]

\[ | \lambda x. M \]

\[ | x \]

\[ | \text{thunk } M \]

\[ | \text{force } V \]

\[ | \text{return } V \]

\[ | \text{returner } M_1 \text{ to } x. M_2 \]

\[ \overline{C, D} ::= \ldots \]

\[ | A \rightarrow C \]

\[ | FA \]

Computation types:

Value terms:

\[ V, W ::= c | \ldots \]

\[ \text{constants, products, etc.} \]

Computation terms:

\[ M, N ::= \ldots \]

\[ \text{products, etc.} \]

\[ | \lambda x. M \]

\[ | V \rightarrow M \]

\[ | \text{functions} \]

\[ | \text{returners} \]
Call-by-push-value syntax

Value types:
\[ A, B ::= \ldots \]
\[ \mid U \subset C \]

Computation types:
\[ C, D ::= \ldots \]
\[ \mid A \to C \]
\[ \mid F A \]

Value terms:
\[ V, W ::= c \mid \ldots \]
\[ \mid \text{thunk } M \]
\[ \mid x \]
\[ \mid \text{thunks} \]

Computation terms:
\[ M, N ::= \ldots \]
\[ \mid \lambda x. M \mid V \downarrow M \]
\[ \mid \text{return } V \mid M_1 \text{ to } x. M_2 \]
\[ \mid \text{returners} \]
\[ \mid \text{functions} \]

\[ \Gamma \vdash M : C \]
\[ \Gamma \vdash V : U \subset C \]
\[ \Gamma \vdash \text{thunk } M : U \subset C \]
\[ \Gamma \vdash \text{force } V : C \]
Call-by-push-value syntax

Value types:

\[ A, B ::= \ldots \]
\[
| \mathsf{U} C
\]

Computation types:

\[ C, D ::= \ldots \]
\[
| A \rightarrow C
\]
\[
| FA
\]

Value terms:

\[ V, W ::= c | \ldots \]
\[
| \mathsf{thunk} M
\]
\[
| x
\]

Computation terms:

\[ M, N ::= \ldots \]
\[
| \lambda x. M | V \ ' M
\]
\[
| \mathsf{return} V | M_1 \ \mathsf{to} \ x. M_2
\]
\[
| \mathsf{force} V
\]

Typing contexts:

\[ \Gamma ::= \emptyset | x : A \]
Call-by-push-value syntax

Value types:

\[ A, B ::= \ldots \]

\[ | \text{U} C \]

Value terms:

\[ V, W ::= c \mid \ldots \] constants, products, etc.

\[ | \text{thunk} \ M \] thunks

\[ | x \]

Computation types:

\[ C, D ::= \ldots \]

\[ | A \to C \]

Computation terms:

\[ M, N ::= \ldots \] products, etc.

\[ | \lambda x. \ M \mid V \ \text{`} \ M \] functions

\[ | \text{return} \ V \mid M_1 \ \text{to} \ x. \ M_2 \] returners

\[ | \text{force} \ V \]

\[\frac{\Gamma \vdash V : A}{\Gamma \vdash \text{return} \ V : FA}\]

\[\frac{\Gamma \vdash M_1 : FA \quad \Gamma, x : A \vdash M_2 : C}{\Gamma \vdash M_1 \ \text{to} \ x. \ M_2 : C}\]
Some side-effects

Recursion:

\[
\Gamma, x : \text{U} C \vdash M : C \\
\Gamma \vdash \text{rec } x : \text{U} C. M : C
\]

Nondeterminism:

\[
\Gamma \vdash M_1 : C \quad \Gamma \vdash M_2 : C \\
\Gamma \vdash M_1 \text{ or } M_2 : C
\]
Call-by-value and call-by-name

Source language types:

\[ \tau ::= \text{unit} \mid \text{bool} \mid \tau \to \tau' \]

Translations from value and name into CBPV:

\[ \tau \mapsto \text{value type } (\tau)^V \quad \tau \mapsto \text{computation type } (\tau)^n \]

- \text{unit} \mapsto \text{unit}
- \text{bool} \mapsto \text{bool}
- \tau \to \tau' \mapsto \text{U}(\tau^V \to \text{F}(\tau')^V)
- \tau \to \tau' \mapsto (\text{U}(\tau)^n \to (\tau')^n)

\[ \begin{align*}
\Gamma, x : \tau & \mapsto (\Gamma)^V, x : (\tau)^V \\
\Gamma \vdash e : \tau & \mapsto (\Gamma)^V \vdash (e)^V : \text{F}(\tau)^V \\
\Gamma, x : \tau & \mapsto (\Gamma)^n, x : \text{U}(\tau)^n \\
\Gamma \vdash e : \tau & \mapsto (\Gamma)^n \vdash (e)^n : (\tau)^n
\end{align*} \]
Call-by-value and call-by-name

Want to relate

\[
\frac{\langle \Gamma \rangle^v \quad \langle e \rangle^v}{\Rightarrow} \quad \frac{\langle \tau \rangle^v}{F}
\]

to

\[
\frac{\langle \Gamma \rangle^n \quad \langle e \rangle^n}{\Rightarrow} \quad \frac{\langle \tau \rangle^n}{n}
\]
ON THE RELATION BETWEEN DIRECT AND CONTINUATION SEMANTICS

John C. Reynolds
Systems and Information Science
Syracuse University

ABSTRACT: The use of continuations in the definition of programming languages has gained considerable currency recently, particularly in conjunction with the lattice-theoretic methods of D. Scott. Although continuations are apparently needed to provide a mathematical semantics for non-applicative control features, they are unnecessary for the definition of a purely applicative language, even when call-by-value occurs. This raises the question of the relationship between the direct and the continuation semantic functions for a purely applicative language. We give two theorems which specify this relationship and show that, in a precise sense, direct semantics are included in continuation semantics.

The heart of the problem is the construction of a relation which must be a fixed-point of a non-monotonic "relational functor." A general method is given for the construction of such relations between recursively defined domains.
Call-by-value and call-by-name

Want to relate

\[
\langle \Gamma \rangle^v \quad \langle e \rangle^v \xrightarrow{\text{F}} \quad \langle \tau \rangle^v
\]

to

\[
\langle \Gamma \rangle^v \quad \xrightarrow{\langle e \rangle^n} \quad \langle \Gamma \rangle^n \quad \xrightarrow{\langle \tau \rangle^n} \quad \text{F} \langle \tau \rangle^v
\]
Call-by-value and call-by-name

Define maps between call-by-value and call-by-name computations?

\[ \Gamma \vdash M : F(\tau)^v \iff \Gamma \vdash \Phi_\tau M : (\tau)^n \]
\[ \Gamma \vdash N : (\tau)^n \iff \Gamma \vdash \Psi_\tau N : F(\tau)^v \]
Call-by-value and call-by-name

Define maps between call-by-value and call-by-name computations?

\[ \Gamma \vdash M : F(\tau)^v \quad \mapsto \quad \Gamma \vdash \Phi_\tau M : (\tau)^n \]

\[ \Gamma \vdash N : (\tau)^n \quad \mapsto \quad \Gamma \vdash \Psi_\tau N : F(\tau)^v \]

Example: for \( \tau = \text{unit} \to \text{unit} \)

\( M \) \text{ CBV to CBN} \quad \mapsto \quad M \text{ to } f.\lambda x. \text{ force } x \text{ to } z. z \text{ ‘ force } f \)

\( N \) \text{ CBN to CBV} \quad \mapsto \quad \text{return thunk } \lambda x. (\text{thunk return } x) \text{ ‘ } N \)
Call-by-value and call-by-name

Want to relate

$$
\langle \Gamma \rangle^v \xrightarrow{\langle e \rangle^v} F \langle \tau \rangle^v
$$

to

$$
\langle \Gamma \rangle^v \rightarrow \langle \Gamma \rangle^n \xrightarrow{\langle e \rangle^n} \langle \tau \rangle^n \rightarrow F \langle \tau \rangle^v
$$
Assume some denotational semantics for CBPV:

- If $\Gamma \vdash M : C$ then $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \to \llbracket C \rrbracket$
- Require order-enrichment: $\llbracket M \rrbracket \sqsubseteq \llbracket N \rrbracket$

Examples:

<table>
<thead>
<tr>
<th>$\llbracket \Gamma \rrbracket$</th>
<th>$\llbracket C \rrbracket$</th>
<th>$\llbracket M \rrbracket$</th>
<th>$\llbracket M \rrbracket \sqsubseteq \llbracket N \rrbracket$</th>
</tr>
</thead>
<tbody>
<tr>
<td>No side-effects</td>
<td>set</td>
<td>set</td>
<td>function</td>
</tr>
<tr>
<td>Recursion</td>
<td>cpo</td>
<td>cpo with $\bot$</td>
<td>continuous func.</td>
</tr>
<tr>
<td>Nondeterminism</td>
<td>poset</td>
<td>join-semilattice</td>
<td>monotone func.</td>
</tr>
</tbody>
</table>

(All of these are adequate: if $\llbracket M \rrbracket \sqsubseteq \llbracket N \rrbracket$ then $M \leq_{ctx} N$)
Maps between CBV and CBN, semantically

Interpret $\Phi_\tau$ and $\Psi_\tau$

$$
\Gamma \vdash M : F(\tau)^v \iff \Gamma \vdash \Phi_\tau M : (\tau)^n
$$

$$
\Gamma \vdash N : (\tau)^n \iff \Gamma \vdash \Psi_\tau N : F(\tau)^v
$$

in the denotational semantics

$$
\phi_\tau : \llbracket F(\tau)^v \rrbracket \rightarrow \llbracket (\tau)^n \rrbracket
$$

$$
\psi_\tau : \llbracket (\tau)^n \rrbracket \rightarrow \llbracket F(\tau)^v \rrbracket
$$

Want to show:

$$
\llbracket (e)^v \rrbracket \sqsubseteq \psi_\tau \circ \llbracket (e)^n \rrbracket \circ \phi_\Gamma
$$
Galois connection between CBV and CBN?

If \((\phi_\tau \circ -, \psi_\tau \circ -)\) is a Galois connection, i.e.

\[
\text{id} \sqsubseteq \psi_\tau \circ \phi_\tau \quad \phi_\tau \circ \psi_\tau \sqsubseteq \text{id}
\]

for each \(\tau\), then

\[
\lceil (e)^V \rceil \sqsubseteq \psi_\tau \circ \lceil (e)^n \rceil \circ \phi_\Gamma
\]
Galois connection between CBV and CBN?

These do not hold in all cases!

\[
\text{id} \sqsubseteq \psi_\tau \circ \phi_\tau \quad \phi_\tau \circ \psi_\tau \sqsubseteq \text{id}
\]

- Don’t hold for: exceptions, printing, state (even if read-only)
- Do hold for: no side-effects, recursion, nondeterminism

This is where the side-effects matter
Galois connection between CBV and CBN?

Definition (Thunkable [Führmann ’99])
A computation $\Gamma \vdash M : FA$ is (lax) thunkable if

$$[[ M \to x. \text{return} (\text{thunk} (\text{return} x)) ]] \sqsubseteq [[ \text{return} (\text{thunk} M) ]]$$

- Essentially: we’re allowed to suspend the computation $M$
- Implies $M$ commutes with other computations, is (lax) discardable, (lax) copyable

Lemma
If everything is thunkable, then $(\phi_\tau \circ -, \psi_\tau \circ -)$ is a Galois connection.

(In adjunction models, assumption is $UF\eta_Y \sqsubseteq \eta_{UFY}$)
How to relate call-by-value to call-by-name

If every computation is thunkable then

$$
\llbracket (e)^v \rrbracket \subseteq \psi_\tau \circ \llbracket (e)^n \rrbracket \circ \phi_\Gamma
$$

for each $e$.

And if $e$ is a program then

$$
\llbracket (e)^v \rrbracket \subseteq \llbracket (e)^n \rrbracket
$$
Examples

If $e$ is a program:

- No side-effects: $\langle e \rangle^y$ and $\langle e \rangle^n$ reduce to the same values
- Nontermination: if $\langle e \rangle^y$ reduces to $v$, then so does $\langle e \rangle^n$
- Nondeterminism: if $\langle e \rangle^y$ can reduce to $v$, then $\langle e \rangle^n$ can also reduce to $v$

But this doesn’t prove anything about exceptions, state, ...
What else can we show?

Local restriction on side-effects: what if the language has other side-effects, but $e$ does not?

▶ Effect systems

What about other evaluation orders?

▶ Use the same technique?
Effect systems

Goal: place upper bound on side-effects of computations

- Replace returner types $FA$ with $\langle \varepsilon \rangle A$
- Track effects $\varepsilon \subseteq \Sigma$

$$\Sigma := \{ \text{diverge, get, put, raise, } \ldots \}$$

$$\Omega : \langle \{\text{diverge}\} \rangle A \quad \text{get : } \langle \{\text{get}\} \rangle \text{bool} \quad \ldots$$

- New theorem (for recursion): if $e$ is a closed program with effect $\varepsilon \subseteq \{\text{diverge}\}$ then

$$\langle e \rangle^v \leadsto^* v \quad \Rightarrow \quad \langle e \rangle^n \leadsto^* v$$
Call-by-need and call-by-name

Call-by-need improves on call-by-name by sharing computations:

\[
\text{let } x = 2 + 2 \text{ in } x + x
\]
Call-by-need and call-by-name

Call-by-need improves on call-by-name by sharing computations:

\[
\text{let } x = 2 + 2 \text{ in } x + x
\]
Call-by-need improves on call-by-name by sharing computations:

\[
\text{let } x = 2 + 2 \text{ in } x + x
\]

\[\rightsquigarrow_{\text{need}} \text{ let } x = 4 \text{ in } x + x\]
Call-by-need and call-by-name

Call-by-need improves on call-by-name by sharing computations:

\[
\begin{align*}
\text{let } x &= 2 + 2 \text{ in } x + x \\
\xrightarrow{\text{need}} \text{let } x &= 4 \text{ in } x + x \\
\xrightarrow{\text{need}} \text{let } x &= 4 \text{ in } 4 + x
\end{align*}
\]
Call-by-need improves on call-by-name by sharing computations:

\[
\text{let } x = 2 + 2 \text{ in } x + x
\]

\[\rightsquigarrow\text{need} \quad \text{let } x = 4 \text{ in } x + x\]

\[\rightsquigarrow\text{need} \quad \text{let } x = 4 \text{ in } 4 + x\]
Call-by-need and call-by-name

Call-by-need improves on call-by-name by sharing computations:

\[
\text{let } x = 2 + 2 \text{ in } x + x
\]

\[\leadsto_{\text{need}} \text{let } x = 4 \text{ in } x + x\]

\[\leadsto_{\text{need}} \text{let } x = 4 \text{ in } 4 + x\]

\[\leadsto_{\text{need}} \text{let } x = 4 \text{ in } 4 + 4\]
Call-by-need and call-by-name

Call-by-need improves on call-by-name by sharing computations:

\[
\text{let } x = 2 + 2 \text{ in } x + x
\]

\[
\xrightarrow{\text{need}} \text{let } x = 4 \text{ in } x + x
\]

\[
\xrightarrow{\text{need}} \text{let } x = 4 \text{ in } 4 + x
\]

\[
\xrightarrow{\text{need}} \text{let } x = 4 \text{ in } 4 + 4
\]
Call-by-need and call-by-name

Call-by-need improves on call-by-name by sharing computations:

\[
\begin{align*}
\text{let } x &= 2 + 2 \text{ in } x + x \\
\leadsto_{\text{need}} \quad \text{let } x &= 4 \text{ in } x + x \\
\leadsto_{\text{need}} \quad \text{let } x &= 4 \text{ in } 4 + x \\
\leadsto_{\text{need}} \quad \text{let } x &= 4 \text{ in } 4 + 4 \\
\leadsto_{\text{need}} \quad \text{let } x &= 4 \text{ in } 8
\end{align*}
\]
Call-by-need and call-by-name

Call-by-need improves on call-by-name by sharing computations:

\[
\text{let } x = 2 + 2 \text{ in } x + x
\]

\[
\overset{\text{need}}{\rightarrow} \text{let } x = 4 \text{ in } x + x
\]

\[
\overset{\text{need}}{\rightarrow} \text{let } x = 4 \text{ in } 4 + x
\]

\[
\overset{\text{need}}{\rightarrow} \text{let } x = 4 \text{ in } 4 + 4
\]

\[
\overset{\text{need}}{\rightarrow} \text{let } x = 4 \text{ in } 8
\]

In some cases (e.g. only recursion), call-by-need and call-by-name should be the same
Call-by-need and call-by-name

Call-by-need is hard: “action at a distance”

- Extend CBPV with new construct

\[
M \text{ need } x. N
\]

- Define a call-by-need translation
- Don’t know how to do denotational semantics for call-by-need
  - Kripke logical relations of varying arity [Jung and Tiuryn ’93]

\[
\mathcal{R}[C] \Gamma \subseteq \text{Term}_C^\Gamma \times \text{Term}_C^\Gamma
\]
Overview

How to relate evaluation orders:

1. Translate from source language to intermediate language
2. Define maps between evaluation orders
3. Relate terms:

\[
\llbracket (e)^v \rrbracket \subseteq \psi_{\tau} \circ \llbracket (e)^n \rrbracket \circ \phi_{\Gamma}
\]

- Works for call-by-value, call-by-name
  - Call-by-need by extending CBPV
- Also works for local restrictions on side-effects using an effect system