Variations on a theme: call-by-value and factorization

Beniamino Accattoli

INRIA & LIX, Ecole Polytechnique
Outline

1. Call-by-value $\lambda$-calculus

2. Factorization
1 Call-by-value $\lambda$-calculus

2 Factorization
Plotkin’s *call-by-value* $\lambda$-calculus:

\[
\begin{align*}
t &::= V \mid t \; t \\
V &::= x \mid \lambda x.\; t
\end{align*}
\]

$\beta_v$ rule: \((\lambda x.\; t)\; V \rightarrow_{\beta_v} t[x/V]\)

- Most functional programming languages are CBV.
- Most works on $\lambda$-calculus are call-by-name (CBN).
Plotkin’s calculus is *not satisfactory* for various reasons.

Semantic models do not faithfully reflect bueibdivergence.

Let $\Delta = \lambda x.xx$. Now consider:

$$M = (\lambda x.\Delta) (y\ z) \ \Delta$$

Semantically $M$ should be *divergent*, but it is a $\beta_v$-*normal form*!

Problem studied by Luca *Paolini* and Simona *Ronchi della Rocca* ("call-by-value solvability").

Another problem: the *completeness* of CPS-translations.
\(\lambda\)-calculus and Linear Logic

- \(\lambda\)-calculus can be represented in various ways inside Linear Logic.

- Two main translations:
  1. **Call-by-name**: \((A \Rightarrow B)^n := (!A^n) \multimap B^n\).
  2. **Call-by-value**: \((A \Rightarrow B)^v := !(A^v \multimap B^v)\).

- Both appear in Girard’s seminal paper (1987)

- Girard calls the second **boring**.

- **Sad consequence**: the CBV-translation is *less known and understood*. 
The translations are typed but both can be extended to pure CBN and CBV $\lambda$-calculus by means of recursive types.

**Curious fact:**

$$M = (\lambda x.\Delta) (y z) \Delta$$

diverges when represented in LL Proof-Nets via the CBV translation (which is good).

**Idea:** to extract the calculus corresponding to CBV Proof-Nets.

Relation with Proof-Nets requires explicit substitutions.

But here ES are evaluated in just one shot.
The value-substitution calculus $\lambda_{vsub}$

- Let $L$ be a possibly empty list $[x_1/u_1] \ldots [x_n/u_n]$.

- Define $\lambda_{vsub}$ as:
  
  $$
  t ::= V \mid t t \mid t[x/u]
  $$$$
  V ::= x \mid \lambda x. t
  $$

- Rules:
  
  $$(\lambda x. t)L s \rightarrow_{dB} t[x/s]L
  $$
  $$(t[x/V]L \rightarrow_{sv} t[x/V]L
  $$

- **Note that** $s$ **needs not** to be a **value**.

- **Note that** explicit substitutions can be reduced only if the content is a **value**.

- **Note** the use of distance (**i.e.** $L$).

- $\lambda_{vsub}$ is **confluent**.
Solvability and explicit substitution

- Re-consider the problematic term:

\[ M = (\lambda w. \Delta) (y \ z) \Delta \]

- Now let’s look at it in our new framework:

\[
(\lambda w. \Delta) (y \ z) \Delta \to_{dB} \Delta[w/y \ z] \Delta \\
(\Delta \Delta)[w/y \ z] \to_{dB} (x x)[x/\Delta][w/y \ z] \to_{sv} \to_{sv} \ldots
\]

- \textit{M has no nf!} (which is good)
Herbelin-Zimmerman’s $\lambda_{CBV}$

- There is a similar calculus by Herbelin and Zimmerman, but without distance.
- The syntax is the same, but not the rules:

  $t ::= V \mid t \ t \mid t[x/u]$

  $V ::= x \mid \lambda x.t$

<table>
<thead>
<tr>
<th>Operational rules</th>
<th>Structural rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\lambda x.t) \ s \Rightarrow t[x/s]$</td>
<td>$t[x/u[y/w]] \rightarrow_{let} t[x/u][y/w]$</td>
</tr>
<tr>
<td>$t[x/V] \rightarrow_{let_v} t[x/V]$</td>
<td>$t[x/u] \ w \rightarrow_{let_app} (t \ w)[x/u]$</td>
</tr>
</tbody>
</table>

- **Note that** $s$ needs not to be a value, but:

  - $(\lambda x.t)[y/w] \ s$ is not a $\Rightarrow$ redex.
  - $t[y/V[x/u]]$ is not a $\rightarrow_{let_v}$ redex.

- The **structural rules** become identities on Proof-Nets.
\( \lambda_{vsub} \) is an **equational sub-calculus** of \( \lambda_{CBV} \):

\[
(\lambda x.t)_L \ s \quad \rightarrow_{db} \quad t[x/s]_L
\]

\[
(\lambda x.t)_L \ s \quad \rightarrow^*_{let_{app}} \quad ((\lambda x.t) \ s)_L \quad \Rightarrow \quad t[x/s]_L
\]

\[
t[x/V_L] \quad \rightarrow_{sv} \quad t[x/V]_L
\]

\[
t[x/V_L] \quad \rightarrow^*_{let_{let}} \quad t[x/V]_L \quad \rightarrow_{let_v} \quad t[V/x]_L
\]

**Thus** \( \rightarrow_{\lambda_{vsub}} \subseteq \rightarrow^*_{\lambda_{CBV}} \).
Apparently, $\lambda_{\text{vsub}}$ is \textit{strictly contained} in $\lambda_{\text{CBV}}$.

These rules \textit{cannot be simulated}:

\[
\begin{align*}
  t[x/u[y/w]] & \rightarrow^{\text{let}} t[x/u][y/w] \\
  t[x/u] w & \rightarrow^{\text{let app}} (t w)[x/u]
\end{align*}
\]

But this \textit{is not} quite true...
Structural congruence

Let $\equiv_0$ be the equivalence relation generated by:

\[
\begin{align*}
&t[x/s][y/u] \sim_1 t[y/u][x/s] \quad \text{if } x \notin \text{fv}(u) & y \notin \text{fv}(s) \\
&t u[x/s] \sim_2 (t u)[x/s] \quad \text{if } x \notin \text{fv}(t) \\
&t[x/s] u \sim_3 (t u)[x/s] \quad \text{if } x \notin \text{fv}(u) \\
&t[x/s[y/u]] \sim_4 t[x/s][y/u] \quad \text{if } y \notin \text{fv}(t)
\end{align*}
\]

$\equiv_0$ contains $\lambda_{CBV}$ structural rules:

\[
\begin{align*}
&t[x/u[y/w]] \rightarrow_{let} t[x/u][y/w] \\
&t[x/u] w \rightarrow_{let_app} (t w)[x/u]
\end{align*}
\]

Operational rules: $t \rightarrow_{\lambda_{CBV}} u$ implies $t \rightarrow_{\lambda_{vsub}} u$.

Structural rules: $t \rightarrow_{\lambda_{CBV}} u$ implies $t \equiv_0 u$.

Hence $\rightarrow_{\lambda_{CBV}} \subseteq (\rightarrow_{\lambda_{vsub}} / \equiv_0)$. 
Strong bisimulations

- $\equiv_\circ$ is a strong bisimulation, i.e.:

$$
\begin{align*}
\text{if } t &\equiv_\circ u \text{ and } t \xrightarrow{\lambda_{v_{\text{sub}}} t'} \exists t' \text{ s.t. } \equiv_\circ \\
\text{then } u &\xrightarrow{\lambda_{v_{\text{sub}}} u'} \equiv_\circ & \equiv_\circ &\equiv_\circ
\end{align*}
$$

- **Rewriting modulo** a strong bisimulation preserves confluence and strong normalisation.

- If $t \equiv_\circ u$ then $t$ and $u$ map to the same **Proof-Net**.

- Then they can really be considered as the same **object**.
In $\lambda_{\text{vsub}}$ there is a \textit{good match} between semantics and divergence.

Recent work in collaboration with Luca Paolini (FLOPS 2012).

This work gives an \textit{operational characterization} of CBV-solvability (a semantic notion).

The operational characterization uses crucially \textit{two factorization theorems}.
Outline

1. Call-by-value $\lambda$-calculus

2. Factorization
A system $S$ is **confluent** when:

\[ t \rightarrow^* u_1 \quad \text{and} \quad t \rightarrow^* u_1 \quad \text{implies} \quad \exists v \text{ s.t.} \quad u_2 \rightarrow^* v \]

A system $S$ is **locally confluent** when:

\[ t \rightarrow u_1 \quad \text{and} \quad t \rightarrow u_1 \quad \text{implies} \quad \exists v \text{ s.t.} \quad u_2 \rightarrow^* v \]

**Termination** $\Rightarrow$ Confluence = Local Confluence (Newman’s Lemma).
General Idea

- $\lambda$-calculus has just one rule:

$$(\lambda x.t)\ u \to_{\beta} t[x/u]$$

which does not terminate.

- Explicit substitutions, abstractly:

1. **Creation of substitutions**: $(\lambda x.t)\L u \to_{d_B} t[x/u]$.
2. **Set of rules executing** substitutions: $t[x/u] \to^* t[x/u]$.

- **Key property**: each rule of an ES-calculus terminates.

- So ES-calculi are sort of locally terminating systems, which are globally non-terminating.
New proof technique for confluence.

Prove local confluence of each rule alone.

Termination gives confluence of each rule.

Hindley-Rosen Lemma: if two reductions $\rightarrow_1$ and $\rightarrow_2$ commute:

$$
t \rightarrow_1^* u_1 \quad \text{implies } \exists v \text{ s.t. } \quad t \rightarrow_1^* u_1 \quad \downarrow_2^* \quad u_2 \rightarrow_1^* v$$

and are confluent then $\rightarrow_1 \cup \rightarrow_2$ is confluent.

Prove commutation of each pair of rule.

Termination often reduces commutation to local commutation.
Local termination

So in ES-calculi a *global* property as confluence can be reduced to *local* confluence and *local* commutation.

Surprising: in $\lambda$-calculus confluence do *not* reduce to local confluence.

ES-calculi are *more complex* than $\lambda$-calculus, but local termination provides *new proof techniques*.

Another notion which can be *localized* is factorization.
Standardization

- Termination is about the *existence* of results.
- Confluence is about the *unicity* of results.
- Standardization instead is about *how to compute*.

- It identifies a specific class of reductions to which any other reduction can be transformed by *permuting its steps*.

- It has many important corollaries, in particular it gives a *normalizing strategy* for evaluation.

- Many applications require a simpler form, called *factorization*.
Factorization is a simple form of standardization.

**Head contexts** in λ-calculus:

\[ H ::= [] | \lambda x.H | H t \]

**Head reduction** \( \rightarrow_h \) in λ-calculus is the closure by head contexts \( H \) of:

\[ (\lambda x.t) u \mapsto_\beta t[x/u] \]

**Internal** reduction is the complement of head reduction, i.e. \( \rightarrow_i := \rightarrow_\beta \setminus \rightarrow_h \).

**Factorization theorem:**

Every reduction \( t \rightarrow^* u \) can be re-organized as \( t \rightarrow_h^* \rightarrow_i^* u \).
At first sight factorization is *easy*.

*Local* diagram permutation diagram:

\[
\begin{array}{ccc}
t & \rightarrow_i & u \\
\downarrow h^+ & \Leftarrow & \downarrow h \\
v & \rightarrow^*_i & w
\end{array}
\]

Two *abstract* lemmas, similar to Newman’s, imply the factorization theorem when:

1. \(\rightarrow^+_h\) is composed of *at most one step*, or
2. \(\rightarrow_h\) is *strongly normalizing*. 
Factorization is non-trivial

- Unfortunately, $\rightarrow_{\beta}$ **lacks** both properties.
- The sequence $\rightarrow_{h}^{+}$ can have length $> 1$:
  
  $\begin{array}{c}
  (\lambda x.x \, x) \, (I \, I) \\
  \downarrow_{h} \\
  (I \, I) \, (I \, I)
  \end{array} \rightarrow_{i} 
  \begin{array}{c}
  \left(\lambda x.x \, x\right) \, I
  \end{array}$

- $\rightarrow_{h}$ is **not** strongly normalising:

  $\begin{array}{c}
  (\lambda x.x \, x) \, \lambda x.x \, x \\
  \rightarrow_{h} \\
  (\lambda x.x \, x \, x) \, \lambda x.x \, x \\
  \rightarrow_{h} \ldots
  \end{array}$
Factorization and explicit substitutions

- The basic ES-calculus $\lambda_{\text{sub}}$:

\[
(\lambda x.t)_L s \mapsto_{d_B} t[x/s]_L \\
t[x/u] \mapsto_s t[x/u]
\]

- Define **head contexts** as:

\[
H ::= [\cdot] \mid \lambda x.H \mid H.t \mid H[x/t]
\]

- We get **four** reductions:

<table>
<thead>
<tr>
<th>$\rightarrow_{dB}$</th>
<th>$\rightarrow_i$</th>
<th>$\rightarrow_h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow_{dB}$</td>
<td>$\rightarrow_{dB}i$</td>
<td>$\rightarrow_{dB}h$</td>
</tr>
<tr>
<td>$\rightarrow_s$</td>
<td>$\rightarrow_{si}$</td>
<td>$\rightarrow_{sh}$</td>
</tr>
</tbody>
</table>

- Remember: they all **terminates**.
We get four diagrams:

\[ t \rightarrow_{dB} u \quad t \rightarrow_{s_i} u \]
\[ \downarrow \ast dB \downarrow dBh \quad \downarrow \ast sh \downarrow sh \]
\[ v \rightarrow_{s_i} w \quad v \rightarrow_{dB} w \]

The abstract lemmas get *factorization of each single diagram* (a new abstract lemma is required).

Glueing the obtained *local factorizations* (easy to do) we get the factorization theorem for \( \lambda_{sub} \).
Call-by-value factors with respect to \textit{weak reductions}.

Weak contexts:

\[ W ::= [\cdot] \mid W \ t \mid t \ W \mid W[x/t] \mid t[x/W] \]

Weak reduction \( \rightarrow_w \): closure of the rules by weak contexts.

Same technique gives \textit{factorization}: if \( t \xrightarrow{\lambda_{\text{sub}}}^{*} u \) then
\[ t \xrightarrow{\rightarrow_w}^{*} u. \]

Factorization also with respect to \textit{stratified weak reduction},
defined from \textit{head-weak} contexts \( H[W] \).
The linear substitution calculus $\lambda_{ls}$:

\[(\lambda x. t) L u \rightarrow_{dB} t[x/u] L\]

\[C[x][x/u] \rightarrow_{ls} C[u][x/u]\]

\[t[x/u] \rightarrow_{w} t \quad x \notin fv(t)\]

Head factorization does not hold:

\[x[x/y[y/z]][z/u] \rightarrow_{ls} x[x/z[y/z]][z/u] \rightarrow_{ls} x[x/u[y/z]][z/u]\]

The two steps cannot be permuted.
New notion of head reduction.
We need to refine the notion of head substitution.
Set:

$$H[x][x/u] \leadsto \text{hls} \quad H[u][x/u]$$

Then define \textit{linear head reduction} as $H[\longrightarrow_{\text{dB}}] \cup H[\longrightarrow_{\text{hls}}]$.

The linear substitution calculus enjoys factorization with respect to linear head reduction.

Linear head reduction can be seen as an abstraction of \textit{Krivine Abstract Machine} (Danos and Regnier).
Linear head reduction arises *naturally* and *repeatedly* in the LL literature.

First studied in connection with Proof-Nets (Mascari, Pedicini).

Then in **semantics**: geometry of interaction and game semantics.

Then in connection with the $\pi$-calculus (Mazza) and **differential $\lambda$-calculus** (Ehrhard, Regnier).

Recently it has been shown to induce a measure for **complexity** (Accattoli, Dal Lago).
THANKS!