Normalisation for Dynamic Pattern Calculi

Eduardo Bonelli

22/May/2012
Pattern Calculi

- A direct approach to capture pattern matching features of programming languages and proof assistants
- Key point: abstraction $\lambda x. u$ is generalised to $\lambda p. u$ where $p$ belongs to a set $P$ of patterns
- Must also specify a matching operation $\{ p \triangleleft_\theta s \}$ to execute an application $(\lambda p. u) s$, whenever possible

Some references:

Algebraic patterns (Peyton-Jones), Lambda-calculus with patterns (van Oostrom), Rho-calculi (Cirstea & Kirchner), Curry-Howard for Gentzen Calculi (Cerrito & Kesner), Lambda-calculus with constructors (Arbiser & Miquel & Ríos) Pure Pattern Calculus (PPC) (Jay & Kesner)
The Pure Pattern Calculus

Example

\[ \text{if } b \text{ then } s \text{ else } r \]

In PPC:

\[ (\lambda \text{True}.\lambda\{x\} \widehat{x}.s) \ b \ r \]

where \( x \) is fresh for \( s \)
The Pure Pattern Calculus

Examples – Pattern Polymorphism

Update all the POINTS of a **LIST**:

\[ \text{updPLIST} \ := \quad f \rightarrow \]

\[ \quad \text{nil} \rightarrow \text{nil} \]

\[ \mid (pt \ y) :: z \rightarrow (pt (f \ y)) :: (\text{updPLIST} \ f \ z) \]

Update all the POINTS of a **TREE**:

\[ \text{updPTREE} \ := \quad f \rightarrow \]

\[ \quad \text{nilT} \rightarrow \text{nilT} \]

\[ \mid \text{nodeT} (pt \ y) z w \rightarrow \text{nodeT} (pt (f \ y)) (\text{updPTREE} \ f \ z) (\text{updPTREE} \ f \ w) \]
The Pure Pattern Calculus

Examples – Pattern Polymorphism

Update all the POINTS of a LIST:

\[
\text{updPLIST} := \begin{cases} 
  f & \rightarrow \text{nil} \\
  \text{nil} & \rightarrow \text{nil} \\
  (pt \ y) :: z & \rightarrow (pt (f \ y)) :: (\text{updPLIST} \ f \ z)
\end{cases}
\]

Update all the POINTS of a TREE:

\[
\text{updPTREE} := \begin{cases} 
  f & \rightarrow \text{nilT} \\
  \text{nilT} & \rightarrow \text{nilT} \\
  \text{nodeT} (pt \ y) z w & \rightarrow \text{nodeT} (pt (f \ y)) (\text{updPTREE} \ f \ z) (\text{updPTREE} \ f \ w)
\end{cases}
\]

Update all the POINTS of any DATA STRUCTURE:

\[
\text{updP} := \begin{cases} 
  f & \rightarrow \begin{cases} 
    pt \ z & \rightarrow pt (f \ z) \\
    x y & \rightarrow (\text{updP} \ f \ x) (\text{updP} \ f \ y) \\
    w & \rightarrow w
\end{cases}
\end{cases}
\]
The Pure Pattern Calculus

Examples – Pattern Polymorphism

Update all the **POINTS** of any **DATA STRUCTURE**:

\[
 updP := f \rightarrow \\
 pt \ z \rightarrow pt (f \ z) \\
 x \ y \rightarrow (updP \ f \ x) (updP \ f \ y) \\
 w \rightarrow w
\]

Update all the **SALARIES** of any **DATA STRUCTURE**:

\[
 updS := f \rightarrow \\
 sl \ z \rightarrow sl (f \ z) \\
 x \ y \rightarrow (updS \ f \ x) (updS \ f \ y) \\
 w \rightarrow w
\]
The Pure Pattern Calculus

Examples – Pattern Polymorphism

Update all the **POINTS** of any **DATA STRUCTURE**:

\[
\text{updP} := f \rightarrow pt z \rightarrow pt (f z)
\]

\[
| \quad x y \rightarrow (\text{updP} f x) (\text{updP} f y)
\]

\[
| \quad w \rightarrow w
\]

Update all the **SALARIES** of any **DATA STRUCTURE**:

\[
\text{updS} := f \rightarrow sl z \rightarrow sl (f z)
\]

\[
| \quad x y \rightarrow (\text{updS} f x) (\text{updS} f y)
\]

\[
| \quad w \rightarrow w
\]

Update all possible **DATA** of any **DATA STRUCTURE**:

\[
\text{upd} := x \rightarrow f \rightarrow
\]

\[
| \quad x y \rightarrow f \rightarrow x (f y)
\]

\[
| \quad z u \rightarrow (\text{upd} x f z) (\text{upd} x f u)
\]

\[
| \quad w \rightarrow w
\]
The Pure Pattern Calculus

Examples – Pattern Polymorphism

Update all the POINTS of any DATA STRUCTURE:

\[ \text{updP} := f \rightarrow \]
\[ pt \ z \rightarrow pt \ (f \ z) \]
\[ x \ y \rightarrow (\text{updP} \ f \ x) \ (\text{updP} \ f \ y) \]
\[ w \rightarrow w \]

Update all the SALARIES of any DATA STRUCTURE:

\[ \text{updS} := f \rightarrow \]
\[ sl \ z \rightarrow sl \ (f \ z) \]
\[ x \ y \rightarrow (\text{updS} \ f \ x) \ (\text{updS} \ f \ y) \]
\[ w \rightarrow w \]

Update all possible DATA of any DATA STRUCTURE:

\[ \text{upd} := \hat{x} \rightarrow \hat{f} \rightarrow \]
\[ x \hat{y} \rightarrow x \ (f \ y) \]
\[ \hat{z} \hat{u} \rightarrow (\text{upd} \ x \ f \ z) \ (\text{upd} \ x \ f \ u) \]
\[ \hat{w} \rightarrow w \]
The Power of Pattern Polymorphism

Update all possible **DATA** of any **DATA STRUCTURE**:

\[ \text{upd} := \hat{x} \rightarrow \hat{f} \rightarrow \]

\[ x \hat{y} \rightarrow x (f y) \]

| \hat{z} \hat{u} | \rightarrow (\text{upd} x f z) (\text{upd} x f u) |
| \hat{w} | \rightarrow w |

\[ \text{upd\ pt} \quad \text{“reduces to”} \quad \text{updP} \]
\[ \text{upd\ sl} \quad \text{“reduces to”} \quad \text{updS} \]

Note, however, that arguments of the function \( \text{upd} \) do not need to be constructors....
Syntax of PPC

- Countable set of symbols \( f, g, \ldots, x, y, z \)
- Sets of symbols denoted \( \theta, \phi, \ldots \)

Terms  

\[
t ::= x | \hat{x} | t t | \lambda_\theta t. t
\]

- \( x \) variable
- \( \hat{x} \) matchable
- \( t u \) application
  - \( t \) is the function and \( u \) the argument
- \( \lambda_\theta p.u \) abstraction
  - \( \theta \) is set of binding symbols, \( p \) is the pattern and \( u \) is the body
  - If \( \theta = \emptyset \) we just write \( \lambda p.u \)
  - Sometimes abbreviate \( \lambda_\theta p.u \) with \( p \to^\theta u \)
Binding in PPC

\[ \lambda_\theta \ p.s \]

- A binding symbol \( x \in \theta \) binds
  - matchable occurrences of \( x \) in \( p \) and
  - variable occurrences of \( x \) in \( s \).

- Example

\[ \lambda\{x\} \ x \, \hat{x} \cdot x \, \hat{x} \]
Binding in PPC

\[ \lambda\{x\} \widehat{x}.(\lambda\{y\} \times \widehat{y}.y) \]

- The inner abstraction \( \lambda\{y\} \times \widehat{y}.y \) binds
  - the only occurrence of the matchable \( \widehat{y} \) in the pattern \( x \times \widehat{y} \) and
  - that of the variable \( y \) in the body \( y \);
- the \( x \) in \( x \times \widehat{y} \) is excluded from \( \{y\} \) since it acts as a place-holder in that pattern
- However, the occurrence of \( x \), as well as that of \( \widehat{x} \), are both bound by the outermost \( \lambda\{x\} \)
- \( \lambda \)-abstraction \( \lambda x.t \) can be defined by \( \lambda\{x\} \widehat{x}.t \)
- The identity function \( \lambda\{x\} \widehat{x}.x \) is abbreviated I
Binding in PPC
Revisiting the update example

\[ \text{upd} := \hat{x} \rightarrow \hat{f} \rightarrow \]
\[ x \hat{y} \rightarrow x (f y) \]
\[ \hat{z} \hat{u} \rightarrow (\text{upd} \times f \hat{z}) (\text{upd} \times f \hat{u}) \]
\[ \hat{w} \rightarrow w \]

Using the lambda notation:

\[ \text{upd} := \lambda_{\{x\}} \hat{x}. \lambda_{\{f\}} \hat{f}. \]
\[ \lambda_{\{y\}} x \hat{y}. x (f y) \]
\[ \lambda_{\{z,u\}} \hat{z} \hat{u}. (\text{upd} \times f z) (\text{upd} \times f u) \]
\[ \lambda_{\{w\}} \hat{w}. w \]
Constructors

- Constructors are matchables which are not bound
- They are often denoted in typewriter fonts a, b, c, d, . . . .

Example

\[ \lambda_{\{x, y\}} \hat{x} \hat{y} \ a.y \]

denotes \( \lambda_{\{x, y\}} \hat{x} \hat{y} \hat{z}.y \)

- The distinction between matchables and variables is unnecessary for standard (static) patterns which do not contain free variables.
Binding in PPC

Free variables and free matchables

\[
\begin{align*}
fv(x) & := \{x\} \\
fv(\hat{x}) & := \emptyset \\
fv(tu) & := fv(t) \cup fv(u) \\
fv(\lambda_\theta p.u) & := (fv(u) \setminus \theta) \cup fv(p)
\end{align*}
\]

\[
\begin{align*}
fm(x) & := \emptyset \\
fm(\hat{x}) & := \{x\} \\
fm(tu) & := fm(t) \cup fm(u) \\
fm(\lambda_\theta p.u) & := (fm(p) \setminus \theta) \cup fm(u)
\end{align*}
\]

Note: We consider terms up to alpha-conversion (i.e. up to renaming of bound matchables and variables)

\[
\lambda_{\{x,y\}}x\hat{x}y\hat{y} \cdot x\hat{x}y\hat{y} =_\alpha \lambda_{\{z,y\}}x\hat{z}y\hat{y} \cdot z\hat{x}y\hat{y}
\]
Dynamics of PPC
A first example

\[
\text{elim ::= } \lambda\{x\} \; \tilde{x}. (\lambda\{y\} \; x \; \tilde{y}. y)
\]

Consider

\[
\text{elim } (\lambda\{z\} \; \tilde{z}. \text{cons } z \; \text{nil})
\]

where \text{cons} and \text{nil} denote constructors

\[
(\lambda\{x\} \; \tilde{x}. (\lambda\{y\} \; x \; \tilde{y}. y)) (\lambda\{z\} \; \tilde{z}. \text{cons } z \; \text{nil})
\]
\[
\rightarrow \lambda\{y\} \; (\lambda\{z\} \; \tilde{z}. \text{cons } z \; \text{nil}) \; \tilde{y}. y
\]
\[
\rightarrow \lambda\{y\} \; \text{cons } \tilde{y} \; \text{nil}. y
\]

- Exhibits pattern polymorphism capabilities of PPC allowing to obtain structurally different deconstructors by applying the same term \text{elim} to different arguments
Dynamics of PPC
A first example

- Reduction in the previous example proceeded smoothly
- However, it may fail
- In order to reduce \((\lambda \theta \ p.s)t\) it must be determined whether \(t\) matches \(p\)
- The match of the pattern \(p\) against the argument \(t\) can be fail (among other possible results detailed shortly)

\[
(\lambda_{\{x\}} \ a \ r_1.x)(b \ r_2) \\
(\lambda_{\{x\}} \ a \ r_1 \ b.x)(a \ r_2 \ d)
\]

- We now take a closer look at matching
Substitution and Matching

- **Substitution** $\sigma$: mapping from variables to terms with finite domain $\text{dom}(\sigma)$
  - Notice that data structures and matchable forms are stable by substitution

- **Match** $\mu$: either a substitution or a special constant in the set \{fail, wait\}

- A match is **positive** if it is a substitution; it is **decided** if it is either positive or fail

- **Application of a match** $\mu$ to a term $t$ ($\mu t$)
  - if $\mu$ is a substitution, then it is applied as usual
  - if $\mu = \text{wait}$, then $\mu t$ is undefined
  - if $\mu = \text{fail}$, then $\mu t$ is the identity function $I$
Substitution and Matching

▶ Disjoint union of matches $\mu_1, \mu_2 \ (\mu_1 \cup \mu_2)$
  
  ▶ their union if both $\mu_i$ are substitutions and $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset$
  
  ▶ wait if either of the $\mu_i$ is wait and none is fail
  
  ▶ fail otherwise.

<table>
<thead>
<tr>
<th>$\cup$</th>
<th>$\sigma_1$</th>
<th>fail</th>
<th>wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_2$</td>
<td>$\sigma_1 \cup \sigma_2 / \text{fail}$</td>
<td>fail</td>
<td>wait</td>
</tr>
<tr>
<td>fail</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>wait</td>
<td>wait</td>
<td>fail</td>
<td>wait</td>
</tr>
</tbody>
</table>

▶ Non-sequential nature of PPC

$$\text{fail} \cup \text{wait} = \text{wait} \cup \text{fail} = \text{fail}$$
Matching operation

- The **compound matching operation** takes a term, a set of binding symbols and a pattern and returns a match.
- Defined by applying the following equations in order:

\[
\begin{align*}
\{\hat{x} \triangleleft_\theta t\} & := \{x \rightarrow t\} \quad \text{if } x \in \theta \\
\{\hat{x} \triangleleft_\theta \hat{x}\} & := \{} \quad \text{if } x \not\in \theta \\
\{pq \triangleleft_\theta tu\} & := \{p \triangleleft_\theta t\} \cup \{q \triangleleft_\theta u\} \quad tu, pq \in MF \\
\{p \triangleleft_\theta t\} & := \text{fail} \quad \text{if } p, t \in MF \\
\{p \triangleleft_\theta t\} & := \text{wait} \quad \text{otherwise}
\end{align*}
\]

Where

- **Data-Structures**
  
  \[D ::= \hat{x} \mid D \cdot t\]

- **Matchable-forms (MF)**
  
  \[F ::= D \mid \lambda_\theta \cdot t.t\]
Matching operation

- The **compound matching operation** takes a term, a set of binding symbols and a pattern and returns a match
- Defined by applying the following equations in order

\[
\begin{align*}
\{\hat{x} \triangleleft_{\theta} t\} &:= \{x \rightarrow t\} & \text{if } x \in \theta \\
\{\hat{x} \triangleleft_{\theta} \hat{x}\} &:= \{\} & \text{if } x \notin \theta \\
\{pq \triangleleft_{\theta} tu\} &:= \{p \triangleleft_{\theta} t\} \cup \{q \triangleleft_{\theta} u\} & tu, pq \in \text{MF} \\
\{p \triangleleft_{\theta} t\} &:= \text{fail} & \text{if } p, t \in \text{MF} \\
\{p \triangleleft_{\theta} t\} &:= \text{wait} & \text{otherwise}
\end{align*}
\]

Where

**Data-Structures**

\[ D ::= \hat{x} \mid D \ t \]

**Matchable-forms (MF)**

\[ F ::= D \mid \lambda_{\theta} \ t.\ t \]

We take a closer look at disjoint union of matches
Disjoint union of matches

- **Disjoint union of matches** $\mu_1, \mu_2$ ($\mu_1 \cup \mu_2$)
  - their union if both $\mu_i$ are substitutions and $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset$
  - wait if either of the $\mu_i$ is wait and none is fail
  - fail otherwise.

<table>
<thead>
<tr>
<th>$\cup$</th>
<th>$\sigma_1$</th>
<th>fail</th>
<th>wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_2$</td>
<td>$\sigma_1 \cup \sigma_2$/fail</td>
<td>fail</td>
<td>wait</td>
</tr>
<tr>
<td>fail</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>wait</td>
<td>wait</td>
<td>fail</td>
<td>wait</td>
</tr>
</tbody>
</table>

- **Non-sequential nature of PPC**

  $\text{fail} \cup \text{wait} = \text{wait} \cup \text{fail} = \text{fail}$
Matching operation

\[
\begin{align*}
\{ \hat{x} \triangleleft_\theta t \} & := \{ x \rightarrow t \} && \text{if } x \in \theta \\
\{ \hat{x} \triangleleft_\theta \hat{x} \} & := \{ \} && \text{if } x \notin \theta \\
\{ pq \triangleleft_\theta tu \} & := \{ p \triangleleft_\theta t \} \uplus \{ q \triangleleft_\theta u \} && \text{if } tu, pq \in \text{MF} \\
\{ p \triangleleft_\theta t \} & := \text{fail} && \text{if } p, t \in \text{MF} \\
\{ p \triangleleft_\theta t \} & := \text{wait} && \text{otherwise}
\end{align*}
\]

Note:

- The use of disjoint union in third case restricts compound matching to linear patterns
- \( p \) linear w.r.t. \( \theta \) if for every \( x \) in \( \theta \), the matchable \( \hat{x} \) appears at most once in \( p \)
- necessary to guarantee confluence
Examples of matching operation

- Decided Match: the positive case
  \((\lambda c \hat{x}. d x)(c r) \rightarrow d r\)
  since \(\{c \hat{x} \triangleleft c r\} = \{x/r\}\).

- Decided Match: the failure case
  \((\lambda c \hat{x}. d x)(a r) \rightarrow I\)
  since \(\{c \hat{x} \triangleleft a r\} = \text{fail}\).

- Non-decided match
  \((\lambda c \hat{x}. d x)((\lambda \hat{x}. x a) c)\) is not a redex
  since \(\{c \hat{x} \triangleleft (\lambda \hat{x}. x a) c\} = \text{wait}\) (argument not ready to match)
More examples of matching operation

- \( \{\hat{x}\hat{x} <_\{x\} uv\} \) gives fail because \( \hat{x}\hat{x} \) is not linear;
- \( \{\hat{x}\hat{y} <_\{x,y,z\} uv\} \) gives fail because \( \{x, y, z\} \neq \{x, y\} \);
- \( \{\hat{x} <_{\emptyset} u\} \) gives fail because \( \emptyset \neq \{x\} \);
- \( \{\hat{y} <_{\{x\}} \hat{y}\} \) gives fail because \( \{x\} \neq \emptyset \);
- \( \{\hat{x}\hat{y} <_{\{x\}} u\hat{z}\} \) gives fail because \( \{\hat{y} <_{\{x\}} \hat{z}\} \) is fail;
- \( \{\hat{x}\hat{y} <_{\emptyset} u\hat{z}\} \) gives fail for the same reason.
Semantics of PPC

\[(\lambda_\theta \ p.s)u \mapsto \{p \triangleleft_\theta u\}s, \text{ if } \{p \triangleleft_\theta u\} \text{ is decided}\]

where

- \(\{p \triangleleft_\theta t\}\) is called the result of the matching operation
- it is defined to be the check of \(\{p \triangleleft_\theta t\}\) on \(\theta\)
  - the check of a match \(\mu\) on \(\theta\) is fail if \(\mu\) is a substitution whose domain is not \(\theta\)
  - it is \(\mu\) otherwise.
Examples

\texttt{elim} := \lambda_{\{x\}} \hat{x}. \lambda_{\{y\}} x \hat{y}. y.

- No extraction from the argument:

\[
\texttt{elim} I a \rightarrow (\lambda_{\{y\}} I \hat{y}. y)a \rightarrow (\lambda_{\{y\}} \hat{y}. y)a \rightarrow a
\]
Examples

\[ \text{elim} := \lambda_{\{x\}} \hat{x}. \lambda_{\{y\}} x \hat{y}. y. \]

- No extraction from the argument:
  \[
  \text{elim} I a \rightarrow (\lambda_{\{y\}} I \hat{y}. y) a \rightarrow (\lambda_{\{y\}} \hat{y}. y) a \rightarrow a
  \]

- Extraction of the unique argument from unary constructors:
  \[
  \text{elim} s (s \ 0) \rightarrow (\lambda_{\{y\}} s \hat{y}. y) (s \ 0) \rightarrow 0
  \]
Examples

\text{elim} := \lambda \{ x \} \hat{x}. \lambda \{ y \} x \hat{y}. y.

\begin{itemize}
\item No extraction from the argument:

\begin{align*}
\text{elim} \, l \, a \rightarrow (\lambda \{ y \} \hat{y}. y) a \rightarrow (\lambda \{ y \} \hat{y}. y) a \rightarrow a
\end{align*}

\item Extraction of the unique argument from unary constructors:

\begin{align*}
\text{elim} \, s \, (s \, 0) \rightarrow (\lambda \{ y \} s \hat{y}. y) (s \, 0) \rightarrow 0
\end{align*}

\item Extraction of argument from binary constructors:

\begin{align*}
\text{elim} \, (\lambda \{ z \} \hat{z}. p \, z \, 1) \, (p \, 0 \, 1) & \rightarrow (\lambda \{ y \} (\lambda \{ z \} \hat{z}. p \, z \, 1) \hat{y}. y) (p \, 0 \, 1) \rightarrow (\lambda \{ y \} p \hat{y} \, 1. y) (p \, 0 \, 1) \rightarrow 0 \\
\text{elim} \, (\lambda \{ z \} \hat{z}. p \, 0 \, z) \, (p \, 0 \, 1) & \rightarrow (\lambda \{ y \} (\lambda \{ z \} \hat{z}. p \, 0 \, z) \hat{y}. y) (p \, 0 \, 1) \rightarrow (\lambda \{ y \} p \, 0 \hat{y}. y) (p \, 0 \, 1) \rightarrow 1
\end{align*}
\end{itemize}
In PPC we can encode alternative cases

Given $\lambda \theta p.s$ and a default case $r$ we define $(\lambda \theta p.s) \mid r$

$$(\lambda \theta p.s) \mid r := \lambda_{\{x\}} \hat{x}.(\lambda \theta p.\lambda_{\{y\}} \hat{y}.s) \times (r \ x)$$
Towards normalisation for PPC

What would be a smart normalising strategy for PPC?

\[(\lambda_\theta p.s)t\]

- Clearly, if the matching of \( p \) against \( t \) is decided, then it is a redex and should be selected.
- When it is non-decided?

It is necessary to understand which subterm (\( p, s \) or \( t \)) needs to be evaluated first in order to attain a normal form for the whole term.

<table>
<thead>
<tr>
<th>Example</th>
<th>Reduced to obtain decided match</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda_{x} I (a \hat{x}).x)(a ; c))</td>
<td>pattern</td>
</tr>
<tr>
<td>((\lambda_{x} a \hat{x}.x)(I (a ; c)))</td>
<td>argument</td>
</tr>
<tr>
<td>((\lambda_{x} a (I (b \hat{x})).x)((I ; a) ; b ; y))</td>
<td>pattern and argument</td>
</tr>
</tbody>
</table>
Towards normalisation for PPC

Which redex to pick in $t_1$ below?

$$(\lambda_{\{x\}}\ a\ (b\ \hat{x})\ r_1.r_2)\ (a\ r_3\ (d\ r_4))$$

- Reducing $r_1$ is not necessarily a good idea (e.g. $r_1$ non-terminating and $r_3$ reduces to $d\ t$, for some $t$)
- Neither is reducing $r_3$ (e.g. $r_3$ is non-terminating and $r_1$ reduces to $b\ t$, for some $t$)
- The same happens if we choose $r_4$.

Both pattern and argument must be reduced in order for $t_1$ to become redex due to non-sequential nature of PPC

$$\text{fail} \cup \text{wait} = \text{wait} \cup \text{fail} = \text{fail}$$
Failure of sequentiality for PPC

- Informal definition of sequentiality: given a term of the form $C[r_1, \ldots, r_n]$ where $C$ does not contain any redex and every $r_i$ is a redex, there exists an index $i$ s.t. $r_i$ is a needed redex and the choice of $i$ is independent from $r_1, \ldots, r_n$

- Recall: A redex $r$ in a term $t$ is needed iff every reduction sequence from $t$ to normal form reduces (a residual of) $r$

- Sequentiality fails for PPC as illustrated by $t_1$

- Here is another example

$$t_2 := (\lambda_{\{y\}} a \ b \ c \ \hat{y}.y) \ (a \ (I \ c) \ (I \ b) \ (I \ a))$$
**Failure of sequentiality for PPC**

\[ t_2 := (\lambda_{\{y\}} \ a \ b \ c \ \hat{y}.y) \ (a \ (I \ c) \ (I \ b) \ (I \ a)) \]

Admits at least two different reduction sequences to normal form:

1. \[ t_2 \rightarrow (\lambda_{\{y\}} \ a \ b \ c \ \hat{y}.y)(a \ c \ (I \ b) \ (I \ a)) \rightarrow \text{nf} \]
2. \[ t_2 \rightarrow (\lambda_{\{y\}} \ a \ b \ c \ \hat{y}.y)(a \ (I \ c) \ b \ (I \ a)) \rightarrow \text{nf} \]

Sequentiality fails in PPC because matching may fail for different reasons: none of the redexes in \( t_2 \) is needed since failure of matching can be declared in terms of (only) \( I \ b \) or \( I \ c \).

- Note: Restriction to static patterns doesn’t change matters since non-sequentiality comes from matching failure.
Multisteps

- It is reasonable to define **multistep** reduction strategies
- These reduce a set of redexes in **one go**
- Example of multistep strategies:
  - **Full** strategy: reduce all the redexes at once
  - **Parallel-Outermost** strategy: reduce all the outermost redexes at once
  - **Outermost-fair** strategy: treat all the outermost redexes fairly, i.e. eventually they must be reduced
$S$ Strategy

Motivation

- Rather than selecting the entire set of outermost redexes, select a refinement
- Call a term of the form $(\lambda_\theta \ p.t)u$ a \textit{preredex}
- $S$ focuses on the leftmost-outermost (LO) preredex of a term
  1. If it is a redex (match decided), select it
  2. If it is not a redex (match not decided), then select only the (outermost) redexes in that subterm of the preredex which helps get it “closer” to a decided match

Example

$$(\lambda\{x,y\} \ a \hat{x} (c \hat{y}).y \ x) \ (a \ r_1 \ r_2)$$

- Reducing $r_1$ will not help
- Reducing $r_2$ will (eg. it may reduce to $c \ s_2$ or $d \ s_2$)
$S$ Strategy

Motivation

$S$ may select various redexes:

$$(\lambda_{\{x,y\}} \ a \ (b \ \widehat{x}) \ r_1 \cdot r_2) \ (a \ r_3 \ (d \ r_4))$$

$S$ selects $r_1$ and $r_3$

$\triangleright$ $r_4$ is delayed since $r_1$ is not in matchable form

$S$ defined by a simultaneous structural analysis of both pattern and argument

$$S : \text{term} \rightarrow \text{set of positions}$$

$$SM : \text{term} \times \text{pattern} \rightarrow \text{set of positions}$$
\textbf{S Strategy}

Definition – (1/2)

\[
S(x) := \emptyset \\
S(\bar{x}) := \emptyset \\
S(\lambda \theta p.t) := \begin{cases} 
1S(p) & \text{if } p \not\in \text{NF} \\
2S(t) & \text{if } p \in \text{NF} 
\end{cases} \\
S((\lambda \theta p.t)u) := \{\epsilon\} & \text{if } \{p \triangleleft \theta u\} \text{ decided} \\
S(tu) := \begin{cases} 
1S(t) & \text{if } t \text{ is not an abstraction and } t \not\in \text{NF} \\
2S(u) & \text{if } t \text{ is not an abstraction and } t \in \text{NF} 
\end{cases}
\]

Example

\[
S((\lambda c\bar{x}.s)(c\,r)) = (\lambda c\bar{x}.s)(c\,r)
\]

What if \(\{p \triangleleft \theta u\}\) is not decided in the fifth clause?
Suppose \( \{p \triangleleft_\theta u\} = \text{wait} \) in all clauses of \( S \) below:

\[
\begin{align*}
S((\lambda_\theta p. t)u) &:= 11G \cup 2D & \text{if } SM_\theta(p, u) = \langle G, D \rangle \neq \langle \emptyset, \emptyset \rangle, \\
S((\lambda_\theta p. t)u) &:= 11S(p) & \text{if } SM_\theta(p, u) = \langle \emptyset, \emptyset \rangle, p \notin \text{NF} \\
S((\lambda_\theta p. t)u) &:= 12S(t) & \text{if } SM_\theta(p, u) = \langle \emptyset, \emptyset \rangle, p \in \text{NF}, t \notin \text{NF} \\
S((\lambda_\theta p. t)u) &:= 2S(u) & \text{if } SM_\theta(p, u) = \langle \emptyset, \emptyset \rangle, p \in \text{NF}, t \in \text{NF}
\end{align*}
\]

\[
\begin{align*}
SM_\theta(\hat{x}, t) &:= \langle \emptyset, \emptyset \rangle & \text{if } x \in \theta \\
SM_\theta(\hat{x}, \hat{x}) &:= \langle \emptyset, \emptyset \rangle & \text{if } x \notin \theta \\
SM_\theta(p_1p_2, t_1t_2) &:= \langle 1G_1 \cup 2G_2, 1D_1 \cup 2D_2 \rangle & \text{if } t_1t_2, p_1p_2 \in \text{MF}, \\
SM_\theta(p_i, t_i) &:= \langle G_i, D_i \rangle & \text{if } p \notin \text{MF} \\
SM_\theta(p, t) &:= \langle S(p), \emptyset \rangle & \text{if } p \in \text{MF} \land t \notin \text{MF} \land \\
SM_\theta(p, t) &:= \langle \emptyset, S(t) \rangle & \neg(p = \hat{x} \land x \in \theta)
\end{align*}
\]
Examples

\[
\begin{align*}
SM \quad (\lambda c \hat{x}.s)((\lambda \hat{z}.za)c) &= (\lambda c \hat{x}.s)((\lambda \hat{z}.za)c) \\
SM \quad (\lambda pf f.s)(p(lt)(lf)) &= (\lambda pf f.s)(p(lt)(lf)) \\
SM \quad (\lambda (lf)f.s)(pf) &= (\lambda (lf)f.s)(pf) \\
SM \quad (\lambda c (lf)f.s)(c f(lf)) &= (\lambda c (lf)f.s)(c f(lf))
\end{align*}
\]
Main result about $S$

- The strategy $S$ is normalising
Conclusions related to normalisation for pattern calculi

Other possible approaches to normalisation for pattern calculi:

▶ Mellières theory of neededness could be reformulated (PPC does not enjoy stability axiom)
▶ Use van Oostrom and van Raamsdonk’s translation of PPC to HORS (non constructive approach)
▶ Using the technique of Kennaway

Future work

1. Adapt the proof to other higher-order languages, or better, to a completely abstract setting
2. Study relation with outermost-fair strategies for higher-order rewriting system
3. Develop an interpreter based on $S$
Technical annex – Proof of normalisation
Lemma

Let \( t \notin \text{NF} \). Then \( S(t) \neq \emptyset \), and \( S(t) \) only contains outermost redexes in \( t \)

Additional observations:

1. \( S \) is not outermost fair

   \[
   (\lambda \, c \, x.s) \Omega
   \]

2. \( S \) coincides with leftmost-outermost unless when dealing with a top non-decided match
Towards proving that $S$ is normalising

\[ t_0 = t \xrightarrow{\Delta_0} u \]
\[ S(t_0) \]
\[ t_1 \xrightarrow{\Delta_1} u \]
\[ \vdots \]
\[ t_n \xrightarrow{\Delta_n} u \]
\[ S(t_n) \]
\[ u \xrightarrow{\Delta_{n+1}} u \]

Proof structure:

- Suppose $t(= t_0)$ has a normal form $u$ and let $\Delta_0$ be any multireduction to $u$
- Construct multireduction $t_0 \xrightarrow{S(t_0)} t_1 \xrightarrow{\Delta_1} u$; where $\Delta_1$ is strictly smaller than the original one
- Show that this ordering on multireductions is well-founded
Measure on multiderivations

- Generalization of measure introduced by R. C. Sekar and I. V. Ramakrishnan
- Also used by van Oostrom for normalisation of weakly orthogonal HORS
- Compares multiderivations of the same length
- We write \( \nu(A) \) for the depth of a multistep \( A \)

Given \( \Delta = \Delta[1..n] \) and \( t_{i-1} \xrightarrow{\Delta[i]} t_i \) for all \( i \), we define \( \chi(\Delta, t_0) \) as the \( n \)-tuple

\[
\langle \nu(\Delta[n], t_{n-1}), \ldots, \nu(\Delta[1], t_0) \rangle
\]

the lexicographic order is used to compare (measures of) multireductions
Measure on multiderivations

Define necessary sets of redexes

Attempt to generalize result of SR93
Necessary sets

- A set of redexes $A$ in $t$ is necessary iff $\forall$ reduction $\delta$ from $t$ to normal form $\exists a \in A$ ($a$ depends on $\delta$) s.t. either $a$ or at least one residual of $a$ is contracted in $\delta$.

- Necessary set generalizes needed redex
  - A singleton $\{r\}$ is necessary iff $r$ is needed.
  - Every term has at least one necessary set (the set of all its redexes).

Example

The set $\{I \, t, \, I \, t\}$ is necessary for $(\lambda p \, f \, f \, f)(p \,(I \, t)(I \, t))$, although none of them is needed.

Theorem

*The strategy $S$ computes necessary sets of redexes.*
Consider the following example (van Oostrom)

\[ (\lambda_x \hat{x}. D x^b) (I y)^{a_2}^{a_1} \xrightarrow{B} (\lambda_x \hat{x}. x x) (I y)^{a_2}^{a_1} \]

where \( A := \{a_1, a_2\}, \ B := \{b\} \) and \( D := \lambda_x \hat{x}. x x \).

- The depth of \( A (= A/\mathcal{B}) \) does change
- the set \( A \) admits a development from \( s' \) requiring two contractions of \( I y \), while this is not the case for \( s \)
Gripping

A redex \( b \) grips another redex \( a \) iff

1. \( a \) is not a fail redex
2. \( b > a \)
3. There are occurrences of variables inside \( b \) which are bound by the abstraction of the redex \( a \)

E.g. \((\lambda y.y) \times\) grips \((\lambda x.(\lambda y.y) \times) \ a\)

- A set of redexes \( A \) in \( t \) is non-gripping iff \( \forall \) reduction \( \delta \) from \( t \) to \( u \) none of the residuals of \( A \) grips the redexes in \( u \)
- Residuals of \( A \) can only appear inside fail redexes

Theorem

*The strategy \( S \) computes non-gripping sets of redexes*
Final result

1. Given a reduction strategy $T$ for PPC, if $T$ computes necessary and $T$ is non-gripping sets then $T$ is normalising.

2. Since the strategy $S$ computes necessary and non-gripping sets, then $S$ is normalising for PPC.

Proof:

- Extends work of Sekar-Ramakrishnan for first-order TRS to an (almost) abstract setting
  - Postponement of unnecessary redexes
  - Decreasing measure for the residuals of non-gripping redexes
- Ideas inspired from van Oostrom are used to define the measure
- Only few auxiliary lemmata depends on the language