Avoiding Repetitive Reduction Patterns in Lambda Calculus with `letrec`

Jan Rochel and Clemens Grabmayer

Dept. of Computer Science, and Dept. of Philosophy
NWO-project *Realising Optimal Sharing*
Utrecht University

TF-lunch
8 March 2011

Apportionment of work

C:

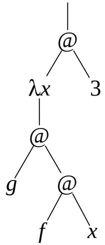
- lambda Calculus; lambda Calculus with `letrec`
- visible and concealed redexes
- optimising `repeat`
- operational equivalence; applicative bisimulation

J:

- generalised β-reduction
- optimising *replicate*
- binding graph
- status quo report; our plans

λ-Calculus

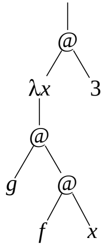
- λ-trees



$(\lambda x.g(fx))3$

λ-Calculus

- λ-trees



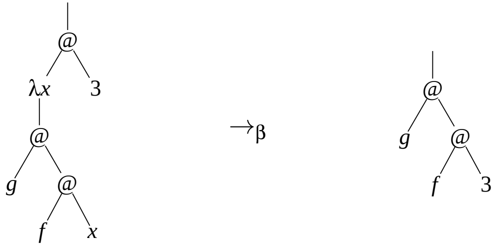
$(\lambda x.g(fx))3$

- λ-terms

$$
\begin{array}{lcll}
T & ::= & V & \text{(variable)} \\
 & | & T\,T & \text{(application)} \\
 & | & \lambda V.T & \text{(abstraction)}
\end{array}
$$

λ-Calculus

- λ-trees



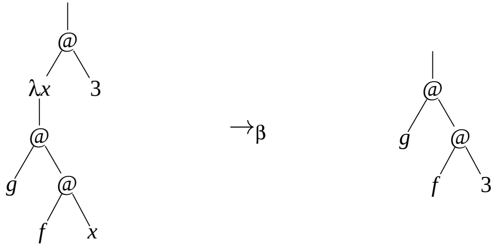
$$(\lambda x.g(fx))3 \quad \to_\beta \quad g(f3)$$

- λ-terms

$$
\begin{array}{lcll}
T & ::= & V & \text{(variable)} \\
  & \mid & T\,T & \text{(application)} \\
  & \mid & \lambda V.T & \text{(abstraction)}
\end{array}
$$

λ-Calculus

- λ-trees



$$(\lambda x.g(fx))3 \quad \to_\beta \quad g(f3)$$

- λ-terms

$$
\begin{array}{rcll}
T & ::= & V & \text{(variable)} \\
 & \mid & T\,T & \text{(application)} \\
 & \mid & \lambda V.T & \text{(abstraction)}
\end{array}
$$

- rewrite rules

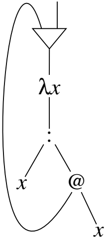
$$(\lambda x.M)N \to M[x := N] \qquad (\beta\textit{-reduction})$$

$$\lambda x.Mx \to M \quad (x \text{ not free in } M) \qquad (\eta\textit{-reduction})$$

λ-Calculus with `letrec`

- λ-graphs

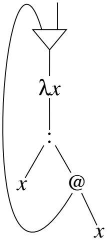
$$r = \lambda x. : (x, (r\, x))$$



λ-Calculus with `letrec`

- λ-graphs

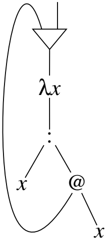
$$r = \lambda x.x : (r\ x)$$



λ-Calculus with `letrec`

- λ-graphs

$$r = \lambda x.x : (r\ x)$$

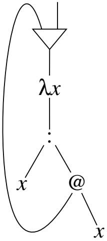


$$\mathbf{let}\ r = \lambda x.x : (r\ x)\ \mathbf{in}\ r$$

λ-Calculus with `letrec`

- λ-graphs

$$r = \lambda x.x : (r\ x)$$

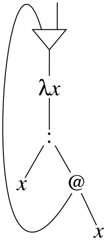


$$\textbf{let}\ r = \lambda x.x : (r\ x)\ \textbf{in}\ r$$

λ-Calculus with `letrec`

- λ-graphs

$r = \lambda x.x : (r\ x)$



$\mathbf{let}\ r = \lambda x.x : (r\ x)\ \mathbf{in}\ r$

- λ_{letrec}-terms (with primitives)

$$
\begin{array}{llll}
T & ::= & V & \text{(variable)} \\
 & \mid & T\ T & \text{(application)} \\
 & \mid & \lambda V . T & \text{(abstraction)} \\
 & \mid & f(T, \ldots, T) & \text{(primitive functions)} \\
 & \mid & \textbf{let}\ \mathit{Defs}\ \textbf{in}\ T & \text{(letrec)} \\
\mathit{Defs} & ::= & v_1 = T\ \ldots\ v_n = T & \text{(equations)} \\
 & & \quad (v_1, \ldots, v_n \text{ distinct variables}) &
\end{array}
$$

mutual recursion

- stream of alternating bits

$$
\begin{array}{l}
\textbf{let} \\
\quad alt = 0 : alt' \\
\quad alt' = 1 : alt \\
\textbf{in} \\
\quad alt
\end{array}
$$

mutual recursion

- stream of alternating bits

$$
\begin{array}{l}
\mathbf{let} \\
\quad alt = 0 : alt' \\
\quad alt' = 1 : alt \qquad \twoheadrightarrow \qquad 0:1:0:1:0:1:\ldots \\
\mathbf{in} \\
\quad alt
\end{array}
$$

mutual recursion

- stream of alternating bits

$$\begin{array}{l}\textbf{let}\\ \quad alt = 0 : alt' \\ \quad alt' = 1 : alt \\ \textbf{in}\\ \quad alt\end{array} \quad \twoheadrightarrow \quad 0:1:0:1:0:1:\ldots$$

- stream of alternating elements

$$\begin{array}{l}\textbf{let}\\ \quad alt = \lambda x.\lambda y.x : (alt'\,x\,y) \\ \quad alt' = \lambda x.\lambda y.y : (alt\,x\,y) \\ \textbf{in}\\ \quad alt\,a\,b\end{array}$$

mutual recursion

- stream of alternating bits

$$\begin{array}{l} \textbf{let} \\ \quad alt = 0 : alt' \\ \quad alt' = 1 : alt \qquad \twoheadrightarrow \qquad 0:1:0:1:0:1:\ldots \\ \textbf{in} \\ \quad alt \end{array}$$

- stream of alternating elements

$$\begin{array}{l} \textbf{let} \\ \quad alt = \lambda x.\lambda y.x : (alt'\, x\, y) \\ \quad alt' = \lambda x.\lambda y.y : (alt\, x\, y) \qquad \twoheadrightarrow \qquad a:b:a:b:a:b:\ldots \\ \textbf{in} \\ \quad alt\ a\ b \end{array}$$

mutual recursion

- Thue–Morse sequence (spec by Larry Moss)

$$
\begin{array}{l}
\textbf{let} \\
\quad L = 0 : X \\
\quad X = 1 : \mathit{zip}\ X\ Y \\
\quad Y = 0 : \mathit{zip}\ Y\ X \\
\quad \mathit{zip}\ (x : \mathit{xs})\ (y : \mathit{ys}) = x : y : \mathit{zip}\ \mathit{xs}\ \mathit{ys} \\
\textbf{in} \\
\quad L
\end{array}
$$

mutual recursion

- Thue–Morse sequence (spec by Larry Moss)

$$
\begin{array}{l}
\textbf{let} \\
\quad L = 0 : X \\
\quad X = 1 : \mathit{zip}\ X\ Y \\
\quad Y = 0 : \mathit{zip}\ Y\ X \\
\quad \mathit{zip}\ (x : xs)\ (y : ys) = x : y : \mathit{zip}\ xs\ ys \\
\textbf{in} \\
\quad L
\end{array}
$$

$$\twoheadrightarrow \quad 0:1:1:0:.:.:.:.:.:.:.:.:.:.:.:.:.:\ldots$$

mutual recursion

- Thue–Morse sequence (spec by Larry Moss)

$$
\begin{array}{l}
\textbf{let} \\
\quad L = 0 : X \\
\quad X = 1 : zip\ X\ Y \\
\quad Y = 0 : zip\ Y\ X \\
\quad zip\ (x : xs)\ (y : ys) = x : y : zip\ xs\ ys \\
\textbf{in} \\
\quad L
\end{array}
$$

$$\twoheadrightarrow \quad 0:1:1:0:1:0:0:1:.:.:.:.:.:.:.:.:\ldots$$

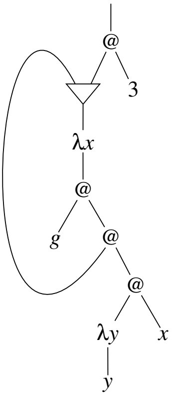
mutual recursion

- Thue–Morse sequence (spec by Larry Moss)

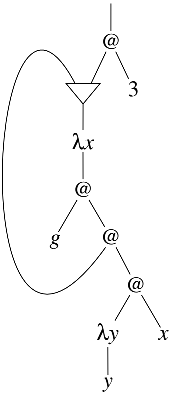
$$
\begin{array}{l}
\textbf{let} \\
\quad L = 0 : X \\
\quad X = 1 : \mathit{zip}\ X\ Y \\
\quad Y = 0 : \mathit{zip}\ Y\ X \\
\quad \mathit{zip}\ (x : xs)\ (y : ys) = x : y : \mathit{zip}\ xs\ ys \\
\textbf{in} \\
\quad L
\end{array}
$$

$$\twoheadrightarrow \quad 0:1:1:0:1:0:0:1:1:0:0:1:0:1:1:0:\ldots$$

Visible and concealed redexes



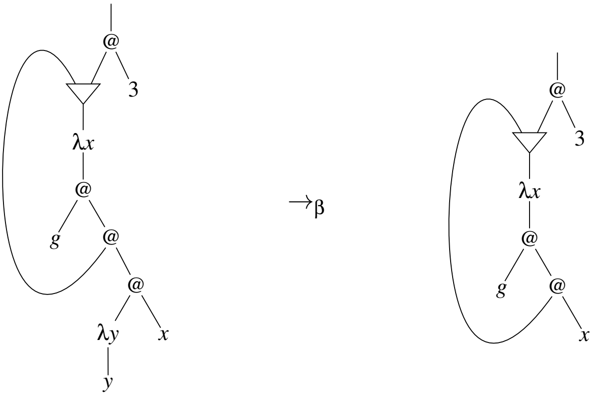
Visible and concealed redexes



In existing compilers:

- visible redexes and their descendants are reduced

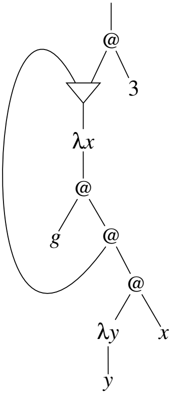
Visible and concealed redexes



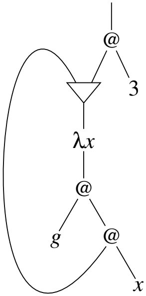
In existing compilers:

- visible redexes and their descendants are reduced

Visible and concealed redexes



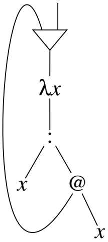
$\to_\beta$



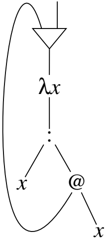
In existing compilers:

- visible redexes and their descendants are reduced
- this does not hold for concealed redexes

repeat

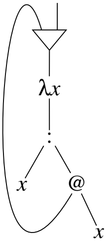


repeat

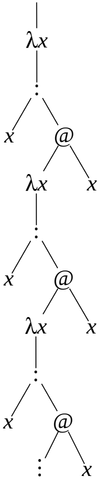


$$\textbf{let } \mathit{repeat} = \lambda x.x : \mathit{repeat}\ x$$
$$\textbf{in } \mathit{repeat}$$

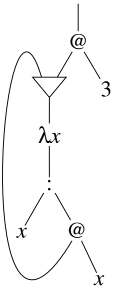
repeat



$$\textbf{let } \mathit{repeat} = \lambda x.x : \mathit{repeat}\ x$$
$$\textbf{in } \mathit{repeat}$$

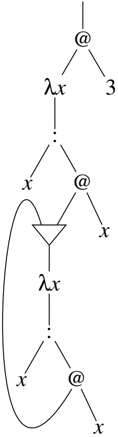


repeat 3

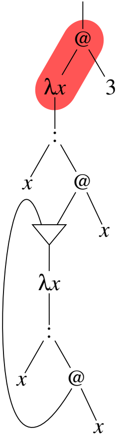


$$\textbf{let } repeat = \lambda x.x : repeat\ x$$
$$\textbf{in } repeat\ 3$$

$\rightarrow_{\triangledown}$

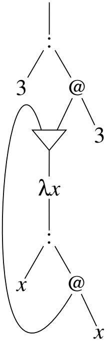


$$\textbf{let } \mathit{repeat} = \lambda x.x : \mathit{repeat}\ x$$
$$\textbf{in } (\lambda x.x : \mathit{repeat}\ x)\ 3$$



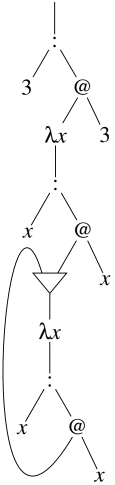
$$\textbf{let}\ repeat = \lambda x.x : repeat\ x$$
$$\textbf{in}\ (\lambda x.x : repeat\ x)\ 3$$

$\rightarrow_\beta$

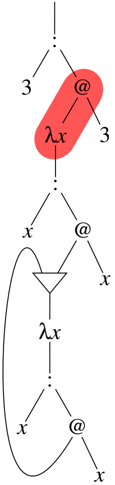


$\textbf{let } \mathit{repeat} = \lambda x.x : \mathit{repeat}\ x$
$\textbf{in } 3 : \mathit{repeat}\ 3$

$\rightarrow_{\triangledown}$

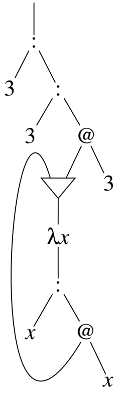


$$\textbf{let } repeat = \lambda x.x : repeat\ x$$
$$\textbf{in } 3 : (\lambda x.x : repeat\ x)\ 3$$



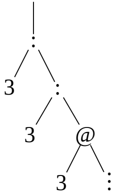
$$\textbf{let } \mathit{repeat} = \lambda x.x : \mathit{repeat}\ x$$
$$\textbf{in } 3 : (\lambda x.x : \mathit{repeat}\ x)\ 3$$

$\rightarrow_\beta$



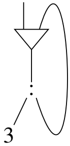
$$\textbf{let}\ repeat = \lambda x.x : repeat\ x$$
$$\textbf{in}\ 3 : 3 : repeat\ 3$$

$\twoheadrightarrow_{\triangledown,\beta}$



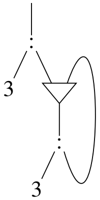
$$3 : 3 : \ldots$$

$\lll\!\!\!-\, \triangledown$



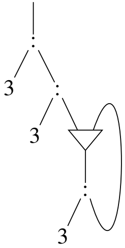
$$\textbf{let}\ rec = 3 : rec\ \textbf{in}\ rec$$

$\rightarrow_{\triangledown}$



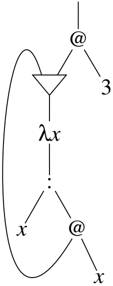
$$\textbf{let}\ rec = 3 : rec\ \textbf{in}\ 3 : rec$$

$\rightarrow_{\triangledown}$

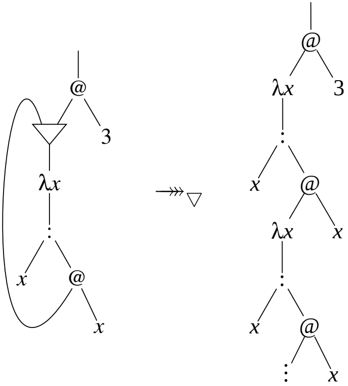


$\textbf{let}\ rec = 3 : rec\ \textbf{in}\ 3 : 3 : rec$

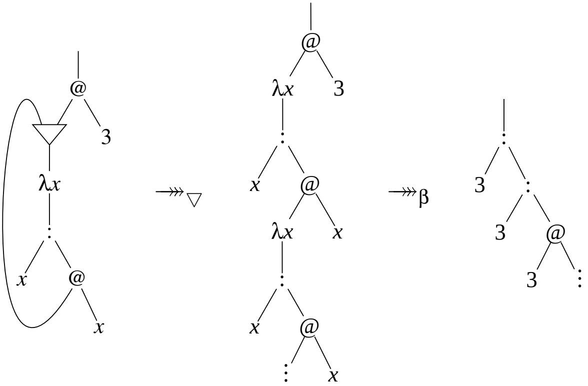
repeat 3



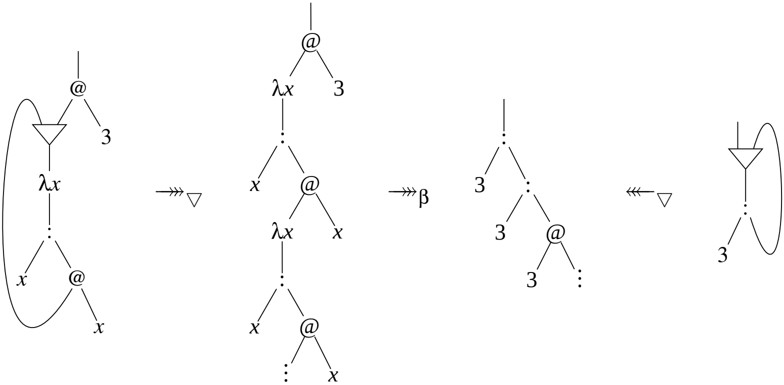
repeat 3



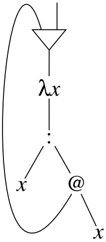
repeat 3



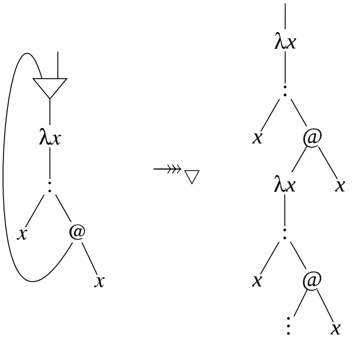
repeat 3



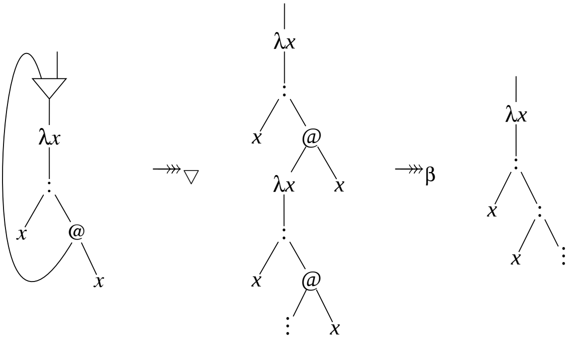
repeat



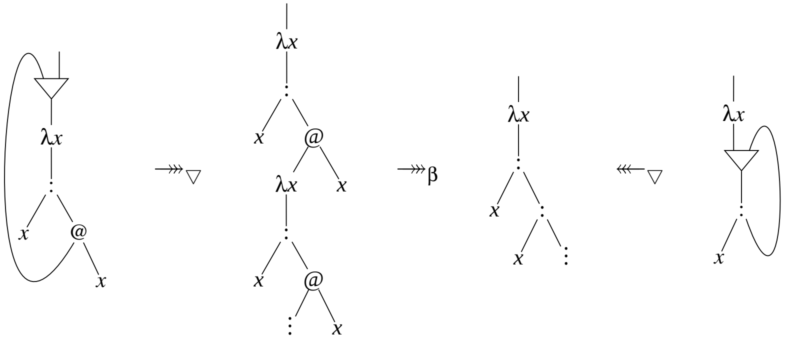
repeat



repeat



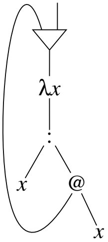
repeat



Optimising *repeat*

(example due to Doaitse Swierstra)

$$\textbf{let } repeat = \lambda x.x : repeat\ x$$
$$\textbf{in } repeat$$



$$=^{\infty}_{\triangledown,\beta}$$

$$repeat = \lambda x.\textbf{let } xs = x : xs \textbf{ in } xs$$

Operational equivalence I

Used here:

$$=^{\infty}_{\triangledown,\beta} = (\twoheadleftarrow_{\triangledown} \cup \twoheadleftarrow_{\beta} \cup \twoheadrightarrow_{\beta} \cup \twoheadrightarrow_{\triangledown})^*$$

as notion of operational equivalence.

Operational equivalence I

Used here:

$$=^{\infty}_{\triangledown,\beta} = (\twoheadleftarrow_{\triangledown} \cup \twoheadleftarrow_{\beta} \cup \twoheadrightarrow_{\beta} \cup \twoheadrightarrow_{\triangledown})^{*}$$

as notion of operational equivalence.

To be shown:

- important observable properties of terms are preserved under $=^{\infty}_{\triangledown,\beta}$.

Applicative bisimulation

Applicative bisimulation (Abramsky, 1990): 'Meaning' of λ-terms defined as repeated evaluation to weak head normal form (as in functional languages).

λ-terms M, N are applicative bisimular ($M \sim^{B} N$) if M and N behave the same under all possible series $E_0, E_1, E_2, \ldots$ of experiments:

Applicative bisimulation

Applicative bisimulation (Abramsky, 1990): ‘Meaning’ of λ-terms defined as repeated evaluation to weak head normal form (as in functional languages).

λ-terms M, N are applicative bisimular ($M \sim^{B} N$) if M and N behave the same under all possible series $E_0, E_1, E_2, \ldots$ of experiments:

E_0 Given term P, reduce P to a weak head normal form; obtain as observations:

1. the whnf of P exists, and is an abstraction $\lambda z.P'$
2. the whnf of P does not exists, or is not an abstraction

Applicative bisimulation

Applicative bisimulation (Abramsky, 1990): 'Meaning' of λ-terms defined as repeated evaluation to weak head normal form (as in functional languages).

λ-terms M, N are applicative bisimular ($M \sim^{B} N$) if M and N behave the same under all possible series $E_0, E_1, E_2, \ldots$ of experiments:

E_0 Given term P, reduce P to a weak head normal form; obtain as observations:

1. the whnf of P exists, and is an abstraction $\lambda z.P'$
2. the whnf of P does not exists, or is not an abstraction

E_{n+1} Suppose the outcome of E_n is the abstraction $\lambda z_n.P_n$. Then:

- choose an arbitrary λ-term Q_{n+1}
- carry out the analogous exper. to E_0 for $(\lambda z_n.P_n)Q_{n+1}$

Operational equivalence vs. applicative bisimulation

Proposition

$=^{\infty}_{\nabla,\beta}$ *is contained in* $\sim^{B}$*: for all terms* M, N*,*

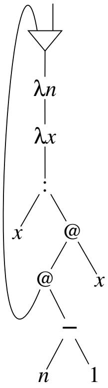
$$M =^{\infty}_{\nabla,\beta} N \quad \Rightarrow \quad M \sim^{B} N$$

replicate

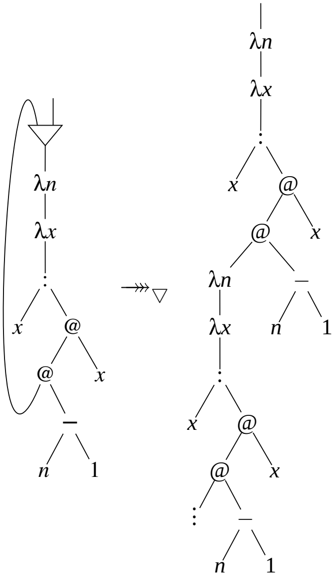
$$
\begin{aligned}
&\mathit{replicate}\ 0\ x = [\,]\\
&\mathit{replicate}\ n\ x = x : \mathit{replicate}\ (n-1)\ x
\end{aligned}
$$

$$
\begin{aligned}
\mathit{replicate}\ n\ x = \mathbf{let}\ &\mathit{rec}\ 0 = [\,]\\
&\mathit{rec}\ n = x : \mathit{rec}\ (n-1)\\
\mathbf{in}\ &\mathit{rec}\ n
\end{aligned}
$$

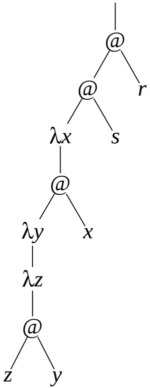
replicate – generalised β-reduction



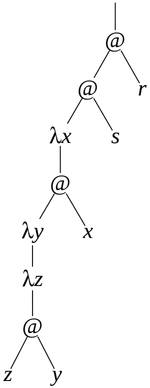
replicate – generalised β-reduction



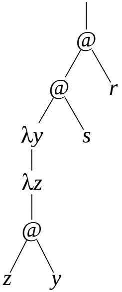
Generalised β-Reduction



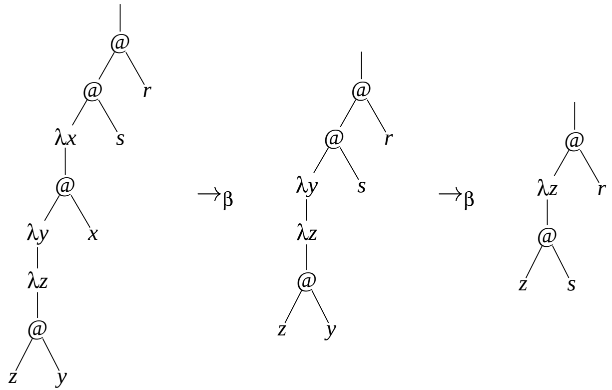
Generalised β-Reduction



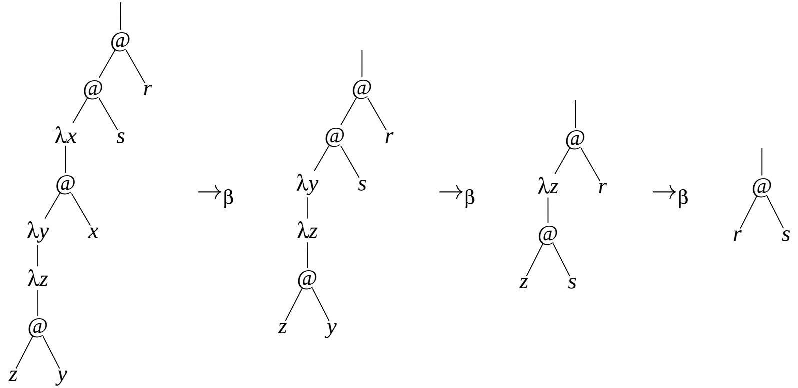
$\to_\beta$



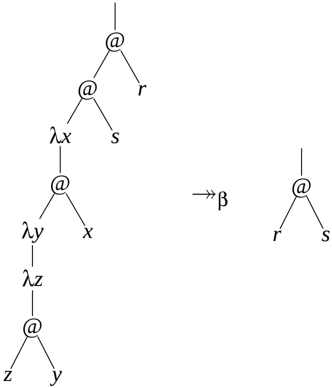
Generalised β-Reduction



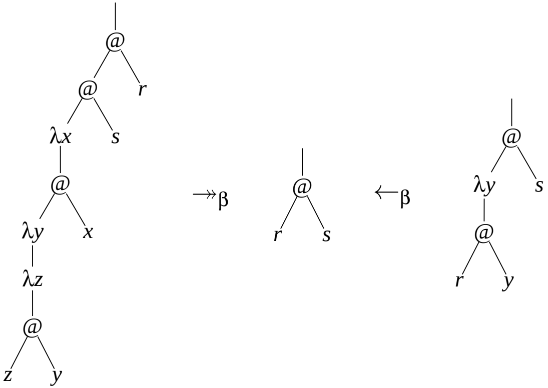
Generalised β-Reduction



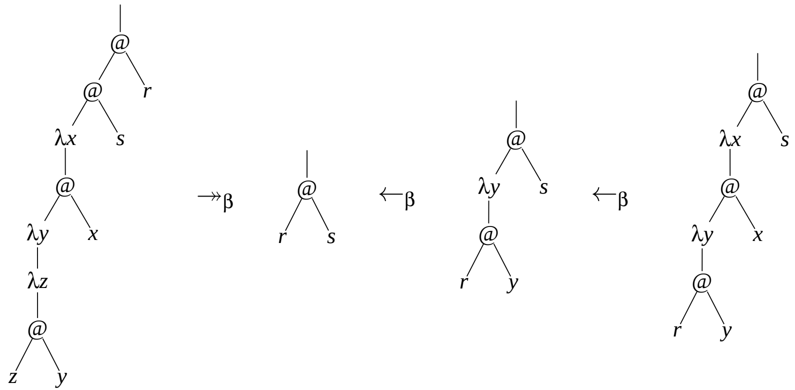
Generalised β-Reduction



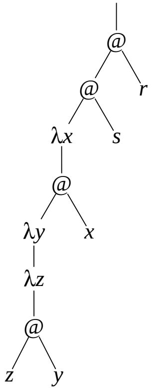
Generalised β-Reduction



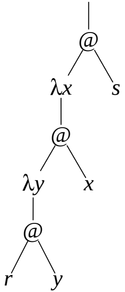
Generalised β-Reduction



Generalised β-Reduction

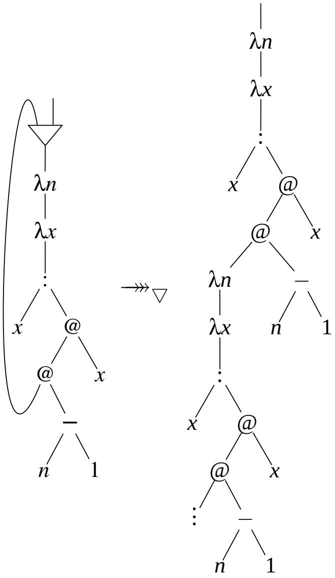


$$\rightarrow_{g\beta}$$

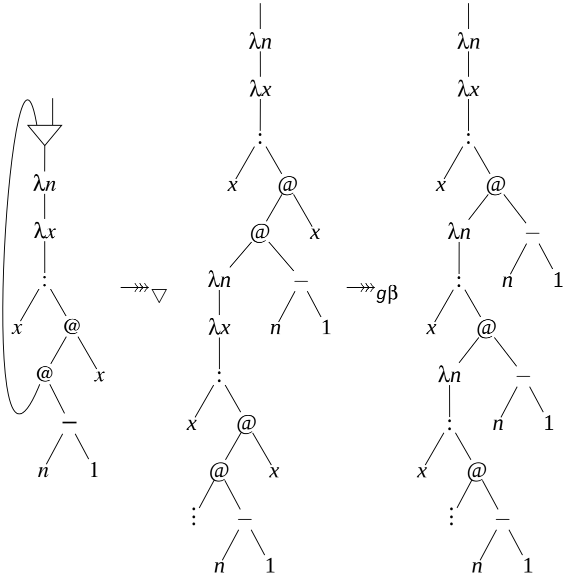


$$\rightarrow_{g\beta} \;\subseteq\; \twoheadrightarrow_{\beta} \cdot \twoheadleftarrow_{\beta}$$

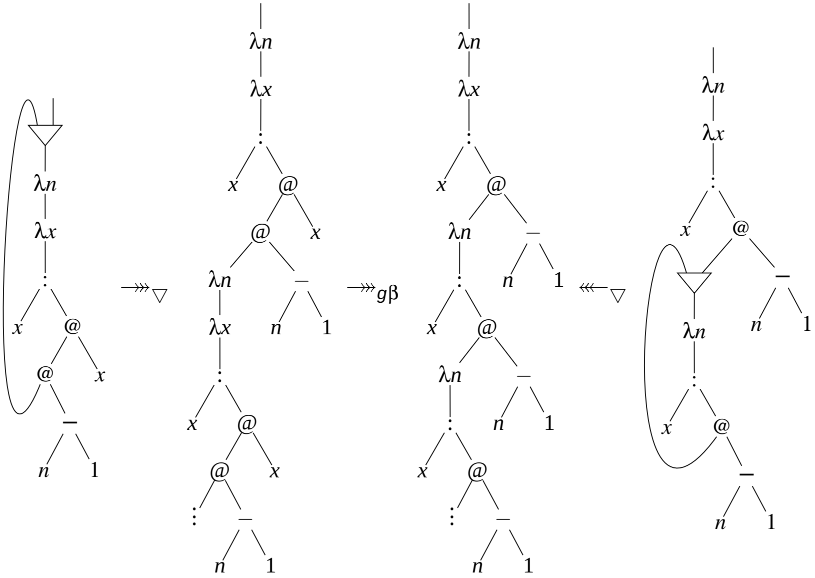
replicate – duplication of the function body



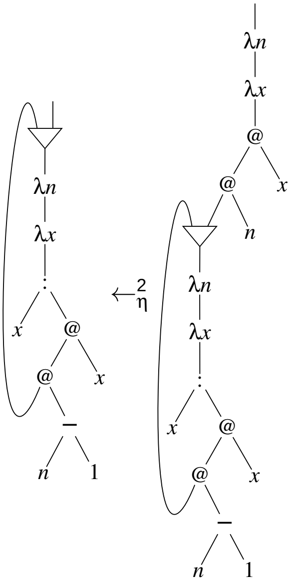
replicate – duplication of the function body



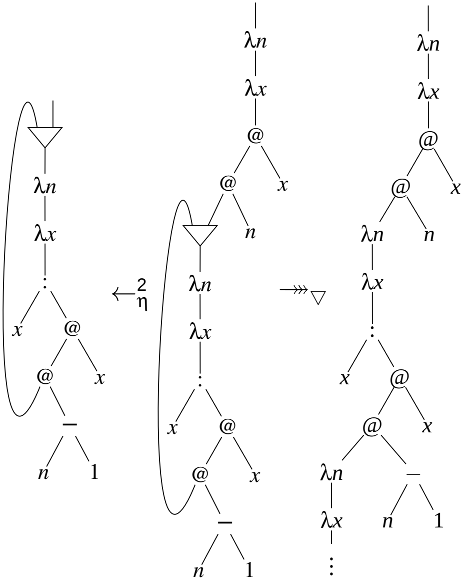
replicate – duplication of the function body



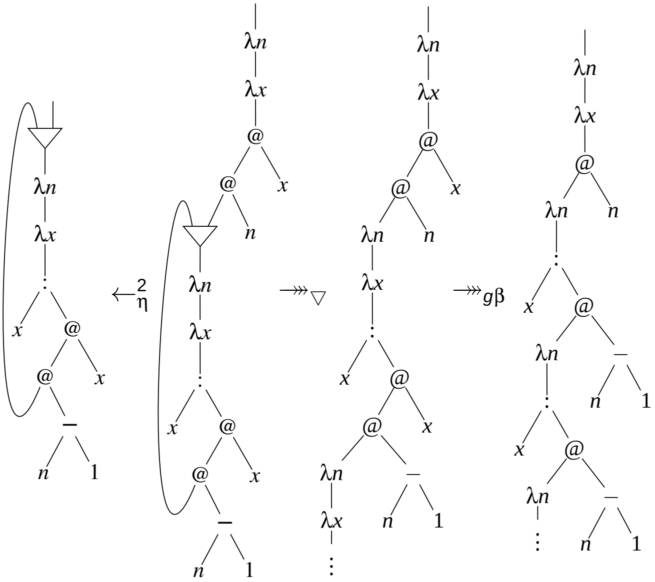
replicate – η-trick



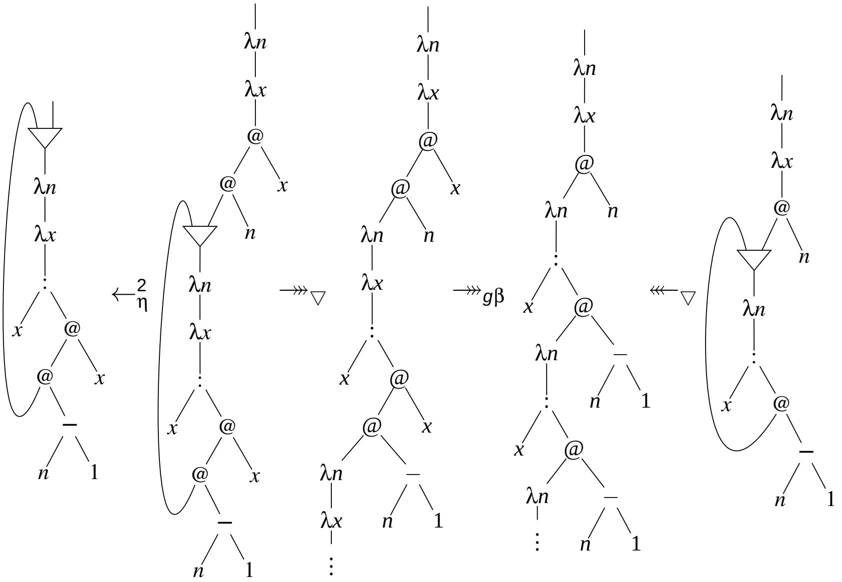
replicate – η-trick



replicate – η-trick



replicate – η-trick



Operational equivalence II

$$=^{\infty}_{\triangledown,g\beta} \quad := \quad \left(\leftarrow_{\eta} \cup \twoheadleftarrow\!\!\leftarrow_{\triangledown} \cup \twoheadleftarrow\!\!\leftarrow_{g\beta} \cup \twoheadrightarrow\!\!\!\rightarrow_{g\beta} \cup \twoheadrightarrow\!\!\!\rightarrow_{\triangledown} \cup \twoheadrightarrow_{\eta} \right)^{*}$$

Proposition

$=^{\infty}_{\triangledown,g\beta} \subseteq \sim^{B}$. Moreover, $=^{\infty}_{\triangledown,g\beta}$ is a refinement of $\sim^{B}$.

Rewrite Rule Formulation

$$f = \lambda x_1. \dots \lambda x_n.\lambda y.C\,[f\ t_1 \dots t_n\ y]$$

$$\rightarrow$$

$$\begin{aligned}
&f = \lambda x_1. \dots \lambda x_n.\lambda y.\\
&\quad \mathbf{let}\ f' = \lambda x_1. \dots \lambda x_n.C\,[f'\ t_1 \dots t_n]\\
&\quad \mathbf{in}\ f'\ x_1 \dots x_n
\end{aligned}$$

Rewriting *repeat*

$$\textbf{let } repeat = \lambda x.x : repeat\ x$$
$$\textbf{in } repeat$$

$$\rightarrow$$

$$repeat = \lambda x.\textbf{let } xs = x : xs \textbf{ in } xs$$

Rewriting *replicate*

$$
\begin{aligned}
&\mathit{replicate}\ 0\ x = [\,]\\
&\mathit{replicate}\ n\ x = x : \mathit{replicate}\ (n-1)\ x
\end{aligned}
$$

$$\rightarrow$$

$$
\begin{aligned}
\mathit{replicate}\ n\ x = \ &\textbf{let}\ \mathit{rec}\ 0 = [\,]\\
&\qquad \mathit{rec}\ n = x : \mathit{rec}\ (n-1)\\
&\textbf{in}\ \mathit{rec}\ n
\end{aligned}
$$

Rewriting *append*

$$
\begin{aligned}
(+\!\!+)\ [\ ]\ ys &= ys \\
(+\!\!+)\ (x : xs)\ ys &= x : xs +\!\!+ ys
\end{aligned}
$$

$$\rightarrow$$

$$
\begin{aligned}
(+\!\!+)\ xs\ ys = \mathbf{let}\ & rec\ [\ ] = ys \\
& rec\ (x : xs) = x : rec\ xs \\
\mathbf{in}\ & rec\ xs
\end{aligned}
$$

Rewriting *map*

$$
\begin{aligned}
&map\ \_\ [\,] &&= [\,]\\
&map\ f\ \ (x : xs) &&= f\ x : map\ f\ xs
\end{aligned}
$$

$$\rightarrow$$

$$
\begin{aligned}
map\ f = \mathbf{let}\ &rec\ [\,] &&= [\,]\\
&rec\ (x : xs) &&= f\ x : rec\ xs\\
\mathbf{in}\ &rec
\end{aligned}
$$

Rewriting *until*

$$until\ p\ f\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ x\ \mathbf{else}\ until\ p\ f\ (f\ x)$$

$$\rightarrow$$

$$\begin{aligned} until\ p\ f\ x = {} & \mathbf{let}\ rec\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ x\ \mathbf{else}\ rec\ (f\ x) \\ & \mathbf{in}\ \ rec\ x \end{aligned}$$

Rewriting *zip*

$$\begin{aligned}
&\mathbf{let}\; x\; a\; b = b : zip\; (x\; a\; b)\; (y\; a\; b)\\
&\quad y\; s\; t\; = s : zip\; (y\; s\; t)\; (x\; s\; t)\\
&\quad zip\; (x : xs)\; (y : ys) = x : y : zip\; xs\; ys\\
&\mathbf{in}\; x
\end{aligned}$$

$$\rightarrow$$

$$\begin{aligned}
&\mathbf{let}\; x\; a\; b = \mathbf{let}\; x' = b : zip\; x'\; (y\; a\; b)\; \mathbf{in}\; x'\\
&\quad y\; s\; t\; = \mathbf{let}\; y' = s : zip\; y'\; (x\; s\; t)\; \mathbf{in}\; y'\\
&\quad zip\; (x : xs)\; (y : ys) = x : y : zip\; xs\; ys\\
&\mathbf{in}\; x
\end{aligned}$$

Binding-Graph Method

$$
\begin{aligned}
\mathbf{let}\ & x\ a\ b = b : zip\ (x\ a\ b)\ (y\ a\ b) \\
& y\ s\ t\ = s : zip\ (y\ s\ t)\ (x\ s\ t) \\
& zip\ (x : xs)\ (y : ys) = x : y : zip\ xs\ ys \\
\mathbf{in}\ & x\ 0\ 1
\end{aligned}
$$

- Binding relation: $\multimap\ \subseteq\ V \times T$
- Binding graph

Strong domination

Strong domination:

$$sdom(v,w)$$

$$\Longleftrightarrow$$

$$v \neq w \;\wedge\; \forall u_0,\ldots,u_n \in V \setminus \{v\}[u_0 \rightarrowtail \ldots \rightarrowtail u_n = w \implies v \rightarrowtail^* u_0 \;\wedge\; u_0 \not\rightarrowtail^* v]$$

Binding-Graph Method

$$
\begin{aligned}
&\textbf{let}\ x\ a\ b = b : zip\ (x\ a\ b)\ (y\ a\ b)\\
&\quad\ \ y\ s\ t\ = s : zip\ (y\ s\ t)\ \ (x\ s\ t)\\
&\quad\ \ zip\ (x : xs)\ (y : ys) = x : y : zip\ xs\ ys\\
&\textbf{in}\ x\ 0\ 1
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let}\ x = 1 : zip\ x\ y\\
&\quad\ \ y = 0 : zip\ y\ x\\
&\quad\ \ zip\ (x : xs)\ (y : ys) = x : y : zip\ xs\ ys\\
&\textbf{in}\ x
\end{aligned}
$$

Domination and the η-trick

$$
\begin{array}{l}
\textbf{let}\; x\; a\; b = b : zip\; (x\; a\; b)\; (y\; a\; b) \\
\quad y\; s\; t\; = s : zip\; (y\; s\; t)\; (x\; s\; t) \\
\quad zip\; (x : xs)\; (y : ys) = x : y : zip\; xs\; ys \\
\textbf{in}\; x
\end{array}
$$

$$
\begin{array}{l}
\lambda a \to \lambda b \to \\
\quad \textbf{let}\; x = b : zip\; x\; y \\
\quad\quad y = a : zip\; y\; x \\
\quad\quad\; zip\; (x : xs)\; (y : ys) = x : y : zip\; xs\; ys \\
\quad \textbf{in}\; x
\end{array}
$$

Current Plans

- upcoming talks
  - TERMGRAPH 2011 (Saarbrücken, 2 April)
  - Computer Science Colloquium (UU)
- practical aspects
  - implementation
  - repetitive reduction patters in the wild: population census
  - benchmarks
  - analysis of effects for different run-time systems
- theoretical aspects
  - formulation of optimisation rewrite rules as HRSs
  - (applicative bisimulation versus η-reduction)
  - domination after unfolding
  - efficiency measure for comparing different results of optimisation
  - interactions between optimisation of different parameter cycles
  - correctness proof
- full paper

Thanks

for your attention!

and for inspiration, and many discussions, to:

- Doaitse Swierstra
- Vincent van Oostrom