Avoiding Repetitive Reduction Patterns in Lambda Calculus with letrec

(Work In Progress)

Jan Rochel and Clemens Grabmayer

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

TERMGRAPH 2011
Saarbrücken, April 2nd
In this talk

We report on:

- an **optimising transformation for λ-calculus with letrec**
- by which i.p. the **cyclic passing on of unchanged arguments during evaluation can often be prevented**
In this talk

We report on:

► an optimising transformation for \( \lambda \)-calculus with \texttt{letrec} \\
► by which i.p. the cyclic passing on of unchanged arguments during evaluation can often be prevented

Examples:

► Haskell functions \texttt{repeat}, \texttt{replicate}, \texttt{++}, \texttt{map}, \texttt{until} \\
► a specification of the Thue–Morse sequence
In this talk

We report on:

- an optimising transformation for $\lambda$-calculus with letrec
- by which i.p. the cyclic passing on of unchanged arguments during evaluation can often be prevented

Examples:

- Haskell functions `repeat`, `replicate`, `++`, `map`, `until`
- a specification of the Thue–Morse sequence

Concepts used:

- visible/concealed redexes
- generalised $\beta$-reduction
- domination in digraphs
- static analysis of cyclically reappearing redexes
\( \lambda \)-Terms and \( \lambda \)-Trees

\[
T ::= \begin{align*}
V & \quad \text{(variable)} \\
T \ T & \quad \text{(application)} \\
\lambda V. \ T & \quad \text{(abstraction)}
\end{align*}
\]
\( \text{\textbf{\( \lambda \)-Terms and \( \lambda \)-Trees}} \)

\[
T ::= \begin{array}{l}
V \quad \text{(variable)}
\\
| \quad T \, T \quad \text{(application)}
\\
| \quad \lambda V. \, T \quad \text{(abstraction)}
\end{array}
\]

\((\lambda x. \, g \, (f \, x)) \, 3\)
\[ (\lambda x. M) N \rightarrow_\beta M[x := N] \]

\[ (\lambda x. g (f \, x)) \, 3 \rightarrow_\beta g (f \, 3) \]
**letrec-Terms and λ-Graphs**

\[
T ::= V \quad \text{(variable)} \\
     T \\ T \quad \text{(application)} \\
     \lambda V . T \quad \text{(abstraction)} \\
     f(T, \ldots, T) \quad \text{(primitive functions)} \\
     \text{let Defs in } T \quad \text{(letrec)} \\
\]

\[
\text{Defs ::= } v_1 = T \ldots v_n = T \quad \text{(equations)} \\
                  (v_1, \ldots, v_n \text{ distinct variables})
\]

```plaintext
let repeat = \lambda x. x: repeat x
in repeat
```
letrec-Terms and $\lambda$-Graphs

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

\text{let } \text{repeat} = \lambda x.\ x : \text{repeat } x \\
\text{in } \text{repeat}
let repeat = \x.x : repeat x
in repeat

\( \text{repeat} \)
let repeat = \x.x : repeat x

in repeat
let repeat = \(x \cdot x : \text{repeat } x\)
in repeat
let repeat = λx.x : repeat x
in repeat
Visible and concealed redexes

Common practice in existing compilers:

- Exhaustive reduction of **visible** redexes
Visible and concealed redexes

Common practice in existing compilers:
- Exhaustive reduction of visible redexes
Visible and concealed redexes

Common practice in existing compilers:

- Exhaustive reduction of visible redexes
- This is in general not possible for concealed redexes
let repeat = \x.x : repeat x
in repeat 3
let \( \text{repeat} = \lambda x. x : \text{repeat} x \) in \( (\lambda x. x : \text{repeat} x) 3 \)
let repeat = \x.x : repeat x
in (\x.x : repeat x) 3
let \( \text{repeat} = \lambda x . x : \text{repeat } x \)

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

\textbf{in} \( 3 : (\lambda x. x : \text{repeat} x) \ 3 \)
\[
\text{let } \text{repeat} = \lambda x. x : \text{repeat } x \\
\text{in } 3 : 3 : \text{repeat } 3
\]
let rec = 3 : rec in rec
let rec = 3 : rec in 3 : rec
let rec = 3 : rec in 3 : 3 : rec
Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$
\textit{repeat 3}
repeat 3
Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$

repeat 3
Rochel, Grabmayer

Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$

\[ \text{repeat} \]
Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$

**repeat**

![Diagram showing the process of avoiding repetitive reduction patterns in $\lambda_{letrec}$](image-url)
Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$

**repeat**

\[ \lambda x : x \xrightarrow{\beta} \lambda x : x \]

\[ \xrightarrow{\triangledown} \]

Diagram: \( \lambda \ x \)
Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$

**repeat**
Optimising \textit{repeat}

\begin{verbatim}
let repeat = \lambda x.x : repeat x
in repeat
\end{verbatim}

\begin{verbatim}
let repeat = \lambda x.let xs = x : xs in xs
in repeat
\end{verbatim}
Operational equivalence I

Used here:

$$\equiv_{\triangle, \beta} = (\leftarrow \triangle \cup \leftarrow \beta \cup \rightarrow \beta \cup \rightarrow \triangle)^*$$

as notion of operational equivalence.
\textit{replicate} \hspace{1cm}

\begin{align*}
\text{replicate} \ 0 \ x &= [] \\
\text{replicate} \ n \ x &= x : \text{replicate} \ (n - 1) \ x
\end{align*}

\begin{align*}
\text{replicate} \ n \ x &= \text{let rec} \ 0 = [] \\
&\hspace{1cm} \text{rec} \ n = x : \text{rec} \ (n - 1) \\
&\hspace{1cm} \text{in rec} \ n
\end{align*}
replicate – generalised β-reduction
replicate – generalised $\beta$-reduction
Generalised β-Reduction
Generalised $\beta$-Reduction
Generalised $\beta$-Reduction

\[ \lambda x \lambda y \lambda z \, s \, r \rightarrow_\beta \lambda y \lambda z \, s \, r \]
Generalised $\beta$-Reduction
Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$

**Generalised $\beta$-Reduction**

\[
\lambda x \ x \rightarrow^\beta \lambda y \ s \rightarrow^\beta \lambda z \ s \rightarrow^\beta \lambda z \ y
\]
Generalised $\beta$-Reduction
Generalised $\beta$-Reduction
Generalised \( \beta \)-Reduction
Generalised $\beta$-Reduction
Generalised $\beta$-Reduction

\[
\lambda x \, s \rightarrow \lambda x \, r \rightarrow g\beta
\]

\[
\lambda y \, x \rightarrow \lambda y \, r \rightarrow g\beta
\]

\[
\lambda z \, y \rightarrow \lambda y \, r \rightarrow \beta \cdot \beta \leftrightarrow \beta \subseteq \leftrightarrow^* \beta
\]
replicate – duplication of the function body
replicate – duplication of the function body
`replicate` – duplication of the function body
replicate – header trick
replicate – header trick
replicate – header trick
replicate – header trick
Operational equivalence II

$g\beta$-Convertibility:

$$\equiv_{\nabla, g\beta} := \left( \leftarrow \nabla \cup \leftarrow g\beta \cup \rightarrow g\beta \cup \rightarrow \nabla \right)^*$$
Rewrite Rule Formulation

\[ f = \lambda x_1 \ldots \lambda x_n \cdot \lambda y. C \left[ f \ t_1 \ldots t_n \ y \right] \]

\[ \rightarrow \]

\[ f = \lambda x_1 \ldots \lambda x_n \cdot \lambda y. \]
\[ \textbf{let} f' = \lambda x_1 \ldots \lambda x_n \cdot C \left[ f' \ t_1 \ldots t_n \right] \]
\[ \textbf{in} f' \ x_1 \ldots x_n \]
Rewriting \( \text{repeat} \)

\[
\text{let } \text{repeat} = \lambda x. x : \text{repeat} \ x \\
\text{in } \text{repeat}
\]

\[
\rightarrow
\]

\[
\text{let } \text{repeat} = \lambda x. \text{let } xs = x : xs \ \text{in } xs \\
\text{in } \text{repeat}
\]
Rewriting *replicate*

\[
\text{replicate } 0 \, x = [] \\
\text{replicate } n \, x = x : \text{replicate} \, (n - 1) \, x \\
\rightarrow \\
\text{replicate } n \, x = \textbf{let} \, \text{rec } 0 = [] \\
\quad \text{rec } n = x : \text{rec} \, (n - 1) \\
\quad \textbf{in} \, \text{rec } n
\]
Rewriting \textit{append}

\[(++ \) \textit{[]} \] \(ys = ys\)
\[(++ \) \((x:xs)\) \] \(ys = x:xs ++ ys\)

\[\rightarrow\]
\[(++ \) \textit{xs} \textit{ys} = \textbf{let rec} \textit{[]} = ys\]
\[\textit{rec} \ (x:xs) = x: rec \ xs\]
\textbf{in} \textit{rec} \textit{xs}\]
Rewriting \textit{map}

\[
\begin{align*}
\text{map } & \quad [ ] \quad = \quad [ ] \\
\text{map } f \ (x : xs) & = \quad f \ x : \text{map } f \ xs \\
\end{align*}
\]

\[
\rightarrow
\]

\[
\begin{align*}
\text{map } f & = \quad \text{let } \text{rec } [ ] \quad = \quad [ ] \\
& \quad \text{rec } (x : xs) = \quad f \ x : \text{rec } xs \\
& \quad \textbf{in } \text{rec}
\end{align*}
\]
Rewriting $\text{until}$

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

$$\rightarrow$$

$$\text{until } p \ f \ x = \text{let rec } x = \text{if } p \ x \ \text{then } x \ \text{else } \text{rec } (f \ x)$$

$$\text{in } \text{rec } x$$
Rewriting the Thue-Morse Sequence

\[
\textbf{let } x \ a \ b = b : \text{zip } (x \ a \ b) \ (y \ a \ b) \\
y \ s \ t = s : \text{zip } (y \ s \ t) \ (x \ s \ t) \\
\text{zip } (x : xs) \ (y : ys) = x : y : \text{zip } xs \ ys \\
\textbf{in } x \ 0 \ 1
\]

\[
\rightarrow
\]

\[
\textbf{let } x \ a \ b = \textbf{let } x' = b : \text{zip } x' \ (y \ a \ b) \ \textbf{in } x' \\
y \ s \ t = \textbf{let } y' = s : \text{zip } y' \ (x \ s \ t) \ \textbf{in } y' \\
\text{zip } (x : xs) \ (y : ys) = x : y : \text{zip } xs \ ys \\
\textbf{in } x \ 0 \ 1
\]
Binding-Graph Method

\[
\text{let } x \ a \ b = b : \text{zip} (x \ a \ b) (y \ a \ b)
\]
\[
y \ s \ t = s : \text{zip} (y \ s \ t) (x \ s \ t)
\]
\[
\text{zip} (x : x s) (y : y s) = x : y : \text{zip} x s y s
\]
\[
in \ x \ 0 \ 1
\]

Binding relation:  \[ \leadsto \subseteq S \times S \]
Binding-Graph Method

\[ \text{let } \mathbf{a,b} = \mathbf{b:zip (x,a,b) (y,a,b)} \]
\[ \mathbf{y,s,t} = \mathbf{s:zip (y,s,t) (x,s,t)} \]
\[ \mathbf{zip (x:xs) (y:ys) = x:y:zip xs ys} \]
\[ \text{in } x\,0\,1 \]

Binding relation: \( \circ \subseteq S \times S \)

Diagram:

- Nodes: 0, 1, a, b, s, t
- Edges: 0 → a → b → 1, s → t
Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$

**Binding-Graph Method**

```
let x a b = b \cdot z i p (x a b) (y a b)

y s t = s \cdot z i p (y s t) (x s t)

z i p (x : x s) (y : y s) = x : y : z i p x s y s

i n x 0 1
```

Binding relation: $\cdot \subseteq S \times S$

![Diagram](image)
Binding-Graph Method

\[
\text{let } x \ a \ b = b : \text{zip} (x \ a \ b) (y \ a \ b) \\
y \ s \ t = s : \text{zip} (y \ s \ t) (x \ s \ t) \\
\text{zip} (x : xs) (y : ys) = x : y : \text{zip} xs ys \\
\text{in } x \ 0 \ 1
\]

Binding relation: \( \circ \subseteq S \times S \)
Binding-Graph Method

\[
\begin{align*}
\text{let } & x \ a \ b = b : \text{zip} \ (x \ a \ b) \ (y \ a \ b) \\
& y \ s \ t = s : \text{zip} \ (y \ s \ t) \ (x \ s \ t) \\
& \text{zip} \ (x : xs) \ (y : ys) = x : y : \text{zip} \ xs \ ys \\
in & x \ 0 \ 1
\end{align*}
\]

Binding relation:  \( \bigcirc \subseteq S \times S \)
Binding-Graph Method

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

Binding relation: \( \circ \subseteq S \times S \)

\begin{tikzpicture}
    \node (a) at (0,0) {a};
    \node (b) at (1,0) {b};
    \node (s) at (0,-1) {s};
    \node (t) at (1,-1) {t};
    \draw (a) -- (0,1);
    \draw (b) -- (1,1);
    \draw (a) -- (s);
    \draw (b) -- (t);
    \draw (s) -- (t);
    \draw (0,0) -- (0,-1);
    \draw (1,0) -- (1,-1);
    \draw (0,1) -- (0,-1); \\
\end{tikzpicture}
Avoiding Repetitive Reduction Patterns in $\lambda_{letrec}$

**Binding-Graph Method**

```plaintext
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
in x 0 1

Binding relation:  $\subseteq S \times S$
```

Diagram:

```
0 ---- a ---- 1
     |      |
    s    t
```

---

Rochel, Grabmayer
Binding-Graph Method

\[
\textbf{let } x \ a \ b = b : \text{zip} \ (x \ a \ b) \ (y \ a \ b) \\
y \ s \ t = s : \text{zip} \ (y \ s \ t) \ (x \ s \ t) \\
\text{zip} \ (x : x s) \ (y : y s) = x : y : \text{zip} \ x s \ y s
\]

\textbf{in} \ x \ 0 \ 1

Binding relation: \( \bigcirc \subseteq S \times S \)
Binding-Graph Method

\[
\text{let } x \ a \ b = b : \text{zip} \ (x \ a \ b) \ (y \ a \ b) \\
y \ s \ t = s : \text{zip} \ (y \ s \ t) \ (x \ s \ t) \\
\text{zip} \ (x : xs) \ (y : ys) = x : y : \text{zip} \ xs \ ys \\
\text{in } x \ 0 \ 1
\]

Binding relation: $\bigcirc \subseteq S \times S$
Binding-Graph Method

\textbf{let} \ x \ a \ b = b : \text{zip} \ (x \ a \ b) \ (y \ a \ b) \\
y \ s \ t = s : \text{zip} \ (y \ s \ t) \ (x \ s \ t) \\
\text{zip} \ (x : xs) \ (y : ys) = x : y : \text{zip} \ xs \ ys \\
\textbf{in} \ x \ 0 \ 1

Binding relation: \ \\
\begin{array}{c}
0 \quad a \\
\quad b \\
\quad s \\
\quad t \\
1
\end{array} \subseteq S \times S
Binding-Graph Method

\[
\text{let } x \ a \ b = b : \text{zip} (x \ a \ b) (y \ a \ b) \\
y \ s \ t = s : \text{zip} (y \ s \ t) (x \ s \ t) \\
\text{zip} (x : x s) (y : y s) = x : y : \text{zip} x s y s \\
\text{in } x \ 0 \ 1
\]

Binding relation: \( \rightarrow \subseteq S \times S \)
Strong domination:

\[ sdom_G(d, w) := \forall p_0 \rightarrow \ldots \rightarrow p_n = v \]

\[ n \geq 0 \]
Strong domination:

\[ sdom_G(d, w) := \]

\[ \forall p_0 \rightarrow \ldots \rightarrow p_n = v : d \in \{ p_0, \ldots, p_n \} \quad n \geq 0 \]
Strong domination:

\[ sdom_G(d, w) := \forall p_0 \rightarrow \ldots \rightarrow p_n = v : d \in \{p_0, \ldots, p_n\} \lor d \rightarrow^+ p_0 \land p_0 \not\rightarrow^+ d \quad n \geq 0 \]
Optimising the Thue-Morse Sequence

\textbf{let } x \ a \ b = b : \text{zip} \ (x \ a \ b) \ (y \ a \ b) \\
y \ s \ t = s : \text{zip} \ (y \ s \ t) \ (x \ s \ t) \\
\text{zip} \ (x : \text{xs}) \ (y : \text{ys}) = x : y : \text{zip} \ \text{xs} \ \text{ys} \\
\textbf{in } x \ 0 \ 1
Optimising the Thue-Morse Sequence

\[
\text{let } x = 1 : \text{zip } x \ y \\
y = 0 : \text{zip } y \ x \\
\text{zip} \ (x : xs) \ (y : ys) = x : y : \text{zip} \ xs \ ys \\
\text{in } x
\]
Current Plans

- practical aspects
  - implementation
  - repetitive reduction patterns in the wild: population census
  - benchmarks
  - analysis of effects for different run-time systems

- theoretical aspects
  - HRS formulation
  - 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