Confluent Let-Floating

Clemens Grabmayer and Jan Rochel

Dept. of Philosophy
Dept. of Information & Computing Sciences
Utrecht University

IWC 2013

28 June 2013

Motivation

$\boldsymbol{\lambda}_{\text{letrec}}$ as an abstraction & the core of functional languages

1. supercombinator translations of functional programs

   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

Motivation

$\boldsymbol{\lambda}_{\mathsf{letrec}}$ as an abstraction & the core of functional languages

1. supercombinator translations of functional programs

   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

2. optimizations of supercombinator transl. (Danvy, Schulz, 1990ies): converse of lambda-lifting:

   lambda-dropping = block-sinking + parameter dropping

Motivation

$\boldsymbol{\lambda}_{\text{letrec}}$ as an abstraction & the core of functional languages

1. supercombinator translations of functional programs
   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

2. optimizations of supercombinator transl. (Danvy, Schulz, 1990ies):
   converse of lambda-lifting:

   lambda-dropping = block-sinking + parameter dropping

3. term graph interpretations of $\boldsymbol{\lambda}_{\text{letrec}}$-terms (ignore let-bindings)

   for definition of a $\boldsymbol{\lambda}_{\text{letrec}}$-term readback desirable:

   canonical representatives of let-floating/block-sinking equiv. classes

Let-floating

1. supercombinator translations of functional programs
   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

Let-floating

1. supercombinator translations of functional programs
(Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\,+\,y\,x)\,x)\,4$

Let-floating

1. supercombinator translations of functional programs
   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\,+\,y\,x)\,x)\,4$

(partial) supercombinator definition

$Y = \lambda xy.\,+\,y\,x$ $X = \lambda x.\,\mathrm{Y}\,x\,x$
$X\,4$

supercombinator definition

Let-floating

1. supercombinator translations of functional programs

   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\,+\,y\,x)\,x)\,4$

$Y = \lambda xy.\,+\,y\,x$
$(\lambda x.\,Y\,x\,x)\,4$

(partial) supercombinator definition

$Y = \lambda xy.\,+\,y\,x$ $X = \lambda x.\,Y\,x\,x$
$X\,4$

supercombinator definition

Let-floating

1. supercombinator translations of functional programs

   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\,+\,y\,x)\,x)\,4$

$(\lambda x.\,(\mathbf{let}\ f = \lambda y.\,+\,y\,x\ \mathbf{in}\ f\,)\,x)\,4$ (naming a subterm)

$Y = \lambda xy.\,+\,y\,x$
$(\lambda x.\,\mathrm{Y}\,x\,x)\,4$

(partial) supercombinator definition

$Y = \lambda xy.\,+\,y\,x$ $X = \lambda x.\,\mathrm{Y}\,x\,x$
$X\,4$

supercombinator definition

Let-floating

1. supercombinator translations of functional programs
   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\,+\,y\,x)\,x)\,4$

$(\lambda x.\,(\mathbf{let}\ f = \lambda y.\,+\,y\,x\ \mathbf{in}\ f)\,x)\,4$ (naming a subterm)

$(\lambda x.\,(\mathbf{let}\ \mathrm{Y} = \lambda x' y.\,+\,y\,x'\ \mathbf{in}\ \mathrm{Y}\,x\,x)\,x)\,4$ (parameter addition)

$Y = \lambda xy.\,+\,y\,x$
$(\lambda x.\,\mathrm{Y}\,x\,x)\,4$

(partial) supercombinator definition

$Y = \lambda xy.\,+\,y\,x$ $X = \lambda x.\,\mathrm{Y}\,x\,x$
$X\,4$

supercombinator definition

Let-floating

1. supercombinator translations of functional programs
   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\, + \, y\, x)\, x)\, 4$

$(\lambda x.\,(\textbf{let } f = \lambda y.\, + \, y\, x \textbf{ in } f)\, x)\, 4$ (naming a subterm)

$(\lambda x.\,(\textbf{let } \mathrm{Y} = \lambda x' y.\, + \, y\, x' \textbf{ in } \mathrm{Y}\, x)\, x)\, 4$ (parameter addition)

$\textbf{let } \mathrm{Y} = \lambda x y.\, + \, y\, x \textbf{ in } (\lambda x.\, \mathrm{Y}\, x\, x)\, 4$

$Y = \lambda x y.\, + \, y\, x$
$(\lambda x.\, \mathrm{Y}\, x\, x)\, 4$

(partial) supercombinator definition

$Y = \lambda x y.\, + \, y\, x$ $X = \lambda x.\, \mathrm{Y}\, x\, x$
$X\, 4$

supercombinator definition

Let-floating

1. supercombinator translations of functional programs (Hughes, Peyton–Jones, 1980ies)

lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\,+\,y\,x)\,x)\,4$

$(\lambda x.\,(\textbf{let } f = \lambda y.\,+\,y\,x \textbf{ in } f\,)\,x)\,4$ (naming a subterm)

$(\lambda x.\,(\textbf{let } \mathrm{Y} = \lambda x' y.\,+\,y\,x' \textbf{ in } \mathrm{Y}\,x\,)\,x)\,4$ (parameter addition)

$\textbf{let } \mathrm{Y} = \lambda x' y.\,+\,y\,x' \textbf{ in } (\lambda x.\,\mathrm{Y}\,x\,x)\,4$

$= \textbf{let } \mathrm{Y} = \lambda xy.\,+\,y\,x \textbf{ in } (\lambda x.\,\mathrm{Y}\,x\,x)\,4$ (α-conversion)

$Y = \lambda xy.\,+\,y\,x$
$(\lambda x.\,\mathrm{Y}\,x\,x)\,4$

(partial) supercombinator definition

$Y = \lambda xy.\,+\,y\,x$ $X = \lambda x.\,\mathrm{Y}\,x\,x$
$X\,4$

supercombinator definition

Let-floating

1. supercombinator translations of functional programs (Hughes, Peyton–Jones, 1980ies)

lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\,+\,y\,x)\,x)\,4$

$(\lambda x.\,(\mathbf{let}\ f = \lambda y.\,+\,y\,x\ \mathbf{in}\ f\,)\,x)\,4$ (naming a subterm)

$(\lambda x.\,(\mathbf{let}\ \mathrm{Y} = \lambda x' y.\,+\,y\,x'\ \mathbf{in}\ \mathrm{Y}\,x\,)\,x)\,4$ (parameter addition)

$\mathrm{let}\nearrow$ $(\lambda x.\,\mathbf{let}\ \mathrm{Y} = \lambda x' y.\,+\,y\,x'\ \mathbf{in}\ \mathrm{Y}\,x\,x)\,4$ (let-lifting over application)

$\mathbf{let}\ \mathrm{Y} = \lambda x' y.\,+\,y\,x'\ \mathbf{in}\ (\lambda x.\,\mathrm{Y}\,x\,x)\,4$

$= \ \mathbf{let}\ \mathrm{Y} = \lambda xy.\,+\,y\,x\ \mathbf{in}\ (\lambda x.\,\mathrm{Y}\,x\,x)\,4$ (α-conversion)

$Y = \lambda xy.\,+\,y\,x$
$(\lambda x.\,\mathrm{Y}\,x\,x)\,4$

(partial) supercombinator definition

$Y = \lambda xy.\,+\,y\,x$ $X = \lambda x.\,\mathrm{Y}\,x\,x$
$X\,4$

supercombinator definition

Let-floating

1. supercombinator translations of functional programs
   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\, + \, y \, x)\, x)\, 4$

	$(\lambda x.\,(\mathbf{let}\ f = \lambda y.\, + \, y \, x \ \mathbf{in}\ f)\, x)\, 4$	(naming a subterm)
	$(\lambda x.\,(\mathbf{let}\ \mathrm{Y} = \lambda x' y.\, + \, y \, x' \ \mathbf{in}\ \mathrm{Y}\, x)\, x)\, 4$	(parameter addition)
let↗	$(\lambda x.\,\mathbf{let}\ \mathrm{Y} = \lambda x' y.\, + \, y \, x' \ \mathbf{in}\ \mathrm{Y}\, x\, x)\, 4$	(let-lifting over application)
let↗	$(\mathbf{let}\ \mathrm{Y} = \lambda x' y.\, + \, y \, x' \ \mathbf{in}\ \lambda x.\,\mathrm{Y}\, x\, x)\, 4$	(let-lifting over abstraction)
	$\mathbf{let}\ \mathrm{Y} = \lambda x' y.\, + \, y \, x' \ \mathbf{in}\ (\lambda x.\,\mathrm{Y}\, x\, x)\, 4$	
=	$\mathbf{let}\ \mathrm{Y} = \lambda x y.\, + \, y \, x \ \mathbf{in}\ (\lambda x.\,\mathrm{Y}\, x\, x)\, 4$	(α-conversion)

$Y = \lambda x y.\, + \, y \, x$
$(\lambda x.\,\mathrm{Y}\, x\, x)\, 4$

(partial) supercombinator definition

$Y = \lambda x y.\, + \, y \, x$ $X = \lambda x.\,\mathrm{Y}\, x\, x$
$X\, 4$

supercombinator definition

Let-floating

1. supercombinator translations of functional programs
   (Hughes, Peyton–Jones, 1980ies)

   lambda-lifting = parameter addition + let-floating

$(\lambda x.\,(\lambda y.\,+\,y\,x)\,x)\,4$

	$(\lambda x.\,(\textbf{let}\ f = \lambda y.\,+\,y\,x\ \textbf{in}\ f)\,x)\,4$	(naming a subterm)
	$(\lambda x.\,(\textbf{let}\ Y = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ Y\,x)\,x)\,4$	(parameter addition)
$\text{let}\nearrow$	$(\lambda x.\,\textbf{let}\ Y = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ Y\,x\,x)\,4$	(let-lifting over application)
$\text{let}\nearrow$	$(\textbf{let}\ Y = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ \lambda x.\,Y\,x\,x)\,4$	(let-lifting over abstraction)
$\text{let}\nearrow$	$\textbf{let}\ Y = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ (\lambda x.\,Y\,x\,x)\,4$	(let-lifting over application)
$=$	$\textbf{let}\ Y = \lambda x y.\,+\,y\,x\ \textbf{in}\ (\lambda x.\,Y\,x\,x)\,4$	(α-conversion)

$Y = \lambda xy.\,+\,y\,x$
$(\lambda x.\,Y\,x\,x)\,4$

(partial) supercombinator definition

$Y = \lambda xy.\,+\,y\,x$ $X = \lambda x.\,Y\,x\,x$
$X\,4$

supercombinator definition

Contribution and terminology

we develop a rewrite analysis of let-floating:

direction	literature	here		sign
upward/outward	let-floating	let-lifting	let-floating	let↗
downward/inward	block-sinking	let-sinking		let↘

Contribution and terminology

we develop a rewrite analysis of let-floating:

direction	literature	here		sign
upward/outward	let-floating	let-lifting	let-floating	let$\nearrow$
downward/inward	block-sinking	let-sinking		let$\searrow$

introduce let-floating HRSs:

- upward/outward: a let-lifting HRS $\mathbf{R}_{\mathsf{let}\nearrow}$
- downward/inward: a let-sinking HRS $\mathbf{R}^{\mathsf{let}\searrow}$

so that these are terminating

Contribution and terminology

we develop a rewrite analysis of let-floating:

direction	literature	here		sign
upward/outward	let-floating	let-lifting	let-floating	let$\nearrow$
downward/inward	block-sinking	let-sinking		let$\searrow$

introduce let-floating HRSs:

- upward/outward: a let-lifting HRS $\mathbf{R}_{\text{let}\nearrow}$
- downward/inward: a let-sinking HRS $\mathbf{R}^{\text{let}\searrow}$

so that these are terminating

show their confluence by:

- critical pair analysis ($\Rightarrow$ local confluence)
- termination
- Newman's Lemma

Contribution and terminology

we develop a rewrite analysis of let-floating:

direction	literature	here		sign
upward/outward	let-floating	let-lifting	let-floating	let $\nearrow$
downward/inward	block-sinking	let-sinking		let $\searrow$

introduce let-floating HRSs:

- upward/outward: a let-lifting HRS $\mathbf{R}_{\mathsf{let}\nearrow}$
- downward/inward: a let-sinking HRS $\mathbf{R}^{\mathsf{let}\searrow}$

so that these are terminating

show their confluence by:

- critical pair analysis modulo (⇒ local confluence modulo)
- termination
- Newman's Lemma

Let-lifting

Abstractions may block $_{\text{let}}\nearrow$-steps, but not applications:

$$(\lambda x.\,(\textbf{let}\ f = \lambda y.\,+\,y\,x\ \textbf{in}\ f\,)\,x)\,4$$

$$\textbf{let}\ \mathsf{Y} = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ (\lambda x.\,\mathsf{Y}\,x\,x)\,4$$

Let-lifting

Abstractions may block $\xrightarrow{}_{\text{let}}\!\nearrow$-steps, but not applications:

$$(\lambda x.\,(\textbf{let}\ f = \lambda y.\,+\,y\,x\ \textbf{in}\ f)\ x)\ 4$$

$$_{\text{let}}\!\nearrow\ (\lambda x.\,\textbf{let}\ f = \lambda y.\,+\,y\,x\ \textbf{in}\ f\ x)\ 4 \qquad \text{(let-lifting over application)}$$

$$\textbf{let}\ \mathsf{Y} = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ (\lambda x.\,\mathsf{Y}\,x\,x)\ 4$$

Let-lifting

Abstractions may block $\text{let}\!\nearrow$-steps, but not applications:

$$(\lambda x.\,(\textbf{let } f = \lambda y.\,+\,y\,x \textbf{ in } f)\;x)\;4$$

$$\text{let}\!\nearrow\;(\lambda x.\,\textbf{let } f = \lambda y.\,+\,y\,x \textbf{ in } f\;x)\;4 \qquad \text{(let-lifting over application)}$$

$$(\lambda x.\,\textbf{let } \mathsf{Y} = \lambda x' y.\,+\,y\,x' \textbf{ in } \mathsf{Y}\;x\;x)\;4 \qquad \text{(parameter addition)}$$

$$\textbf{let } \mathsf{Y} = \lambda x' y.\,+\,y\,x' \textbf{ in } (\lambda x.\,\mathsf{Y}\;x\;x)\;4$$

Let-lifting

Abstractions may block ${}_{\text{let}}\!\nearrow$-steps, but not applications:

$$
\begin{array}{lll}
 & (\lambda x.\, (\textbf{let } f = \lambda y.\, +\, y\, x \textbf{ in } f)\, x)\, 4 & \\
{}_{\text{let}}\!\nearrow & (\lambda x.\, \textbf{let } f = \lambda y.\, +\, y\, x \textbf{ in } f\, x)\, 4 & (\text{let-lifting over application}) \\
 & (\lambda x.\, \textbf{let } \mathrm{Y} = \lambda x' y.\, +\, y\, x' \textbf{ in } \mathrm{Y}\, x\, x)\, 4 & (\text{parameter addition}) \\
{}_{\text{let}}\!\nearrow & (\textbf{let } \mathrm{Y} = \lambda x' y.\, +\, y\, x' \textbf{ in } \lambda x.\, \mathrm{Y}\, x\, x)\, 4 & (\text{let-lifting over abstraction}) \\
 & \textbf{let } \mathrm{Y} = \lambda x' y.\, +\, y\, x' \textbf{ in } (\lambda x.\, \mathrm{Y}\, x\, x)\, 4 &
\end{array}
$$

Let-lifting

Abstractions may block ${}_{\text{let}}\!\nearrow$-steps, but not applications:

$$(\lambda x.\,(\textbf{let}\ f = \lambda y.\,+\,y\,x\ \textbf{in}\ f)\,x)\,4$$

$${}_{\text{let}}\!\nearrow\ (\lambda x.\,\textbf{let}\ f = \lambda y.\,+\,y\,x\ \textbf{in}\ f\,x)\,4 \qquad \text{(let-lifting over application)}$$

$$(\lambda x.\,\textbf{let}\ \mathsf{Y} = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ \mathsf{Y}\,\not{x}\,x)\,4 \qquad \text{(parameter addition)}$$

$${}_{\text{let}}\!\nearrow\ (\textbf{let}\ \mathsf{Y} = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ \lambda x.\,\mathsf{Y}\,x\,x)\,4 \qquad \text{(let-lifting over abstraction)}$$

$${}_{\text{let}}\!\nearrow\ \textbf{let}\ \mathsf{Y} = \lambda x' y.\,+\,y\,x'\ \textbf{in}\ (\lambda x.\,\mathsf{Y}\,x\,x)\,4 \qquad \text{(let-lifting over application)}$$

Let-lifting rules

Let-lifting HRS $\mathbf{R}_{\mathsf{let}\nearrow}$ with rewrite relation $\mathsf{let}\nearrow$:

$$(\mathsf{let}\nearrow @_0) \quad (\mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_0(\vec{f}))\ E_1 \ \rightarrow\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\ E_1$$

$$(\mathsf{let}\nearrow @_1) \quad E_0\,(\mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_1(\vec{f})) \ \rightarrow\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_0\ E_1(\vec{f})$$

$$(\mathsf{let}\nearrow \lambda) \quad \lambda x.\,\mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g},x)\ \mathbf{in}\ E(\vec{f},\vec{g},x)$$
$$\rightarrow \begin{cases} \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \lambda x.\,E(\vec{f},x) & \text{if } \vec{g} \text{ is empty} \\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \lambda x.\,\mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g},x)\ \mathbf{in}\ E(\vec{f},\vec{g},x) & \end{cases}$$

$$(\mathbf{let\text{-}in}_{\_}\ \mathsf{let}\nearrow) \quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E(\vec{f},\vec{g})$$
$$\rightarrow\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E(\vec{f},\vec{g})$$

$$(\mathbf{let}_{\_}\ \mathsf{let}\nearrow) \quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = \mathbf{let}\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \mathbf{in}\ G(\vec{f},g,\vec{h})\ \mathbf{in}\ E(\vec{f},g)$$
$$\rightarrow\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = G(\vec{f},g,\vec{h}),\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \mathbf{in}\ E(\vec{f},g)$$

Let-lifting rules

Let-lifting HRS $\mathbf{R}_{\mathsf{let}\nearrow}$ with rewrite relation $\rightarrow_{\mathsf{let}\nearrow}$:

$$(_{\mathsf{let}}\nearrow @_0) \quad (\mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_0(\vec{f}))\ E_1 \ \rightarrow\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\ E_1$$

$$\mathsf{app}((\mathsf{let}_n\_\mathsf{in}\,(\vec{y}.\,(x_1(\vec{y}),\ldots,x_n(\vec{y}),z_0(\vec{y})))),z_1)$$
$$\rightarrow\ \mathsf{let}_n\_\mathsf{in}\,(\vec{y}.\,(x_1(\vec{y}),\ldots,x_n(\vec{y}),\mathsf{app}(z_0(\vec{y}),z_1)))$$

$$(_{\mathsf{let}}\nearrow @_1) \quad E_0\,(\mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_1(\vec{f})) \ \rightarrow\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_0\ E_1(\vec{f})$$

$$(_{\mathsf{let}}\nearrow \lambda) \quad \lambda x.\,\mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g},x)\ \mathbf{in}\ E(\vec{f},\vec{g},x)$$

$$\rightarrow \begin{cases} \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \lambda x.\,E(\vec{f},x) & \text{if } \vec{g} \text{ is empty} \\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \lambda x.\,\mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g},x)\ \mathbf{in}\ E(\vec{f},\vec{g},x) & \end{cases}$$

$$(\mathbf{let\text{-}in}_{-\,\mathsf{let}}\nearrow) \quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E(\vec{f},\vec{g})$$
$$\rightarrow\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E(\vec{f},\vec{g})$$

$$(\mathbf{let}_{-\,\mathsf{let}}\nearrow) \quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = \mathbf{let}\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \mathbf{in}\ G(\vec{f},g,\vec{h})\ \mathbf{in}\ E(\vec{f},g)$$
$$\rightarrow\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = G(\vec{f},g,\vec{h}),\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \mathbf{in}\ E(\vec{f},g)$$

Let-lifting rules

Let-lifting HRS $\mathbf{R}_{\text{let}\nearrow}$ with rewrite relation $\to_{\text{let}\nearrow}$:

$$(\text{let}\nearrow @_0) \quad (\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f}))\, E_1 \;\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f})\, E_1$$

$$(\text{let}\nearrow @_1) \quad E_0\,(\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_1(\vec{f})) \;\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0\, E_1(\vec{f})$$

$$(\text{let}\nearrow \lambda) \quad \lambda x.\, \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f},\vec{g},x) \textbf{ in } E(\vec{f},\vec{g},x)$$

$$\to \begin{cases} \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, E(\vec{f},x) & \text{if } \vec{g} \text{ is empty} \\ \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, \textbf{let } \vec{g} = \vec{G}(\vec{f},\vec{g},x) \textbf{ in } E(\vec{f},\vec{g},x) \end{cases}$$

$$(\textbf{let-in}_{-}\, {}_{\text{let}\nearrow}) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in let } \vec{g} = \vec{G}(\vec{f},\vec{g}) \textbf{ in } E(\vec{f},\vec{g})$$

$$\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f},\vec{g}) \textbf{ in } E(\vec{f},\vec{g})$$

$$(\textbf{let}_{-}\, {}_{\text{let}\nearrow}) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f},g),\, g = \textbf{let } \vec{h} = \vec{H}(\vec{f},g,\vec{h}) \textbf{ in } G(\vec{f},g,\vec{h}) \textbf{ in } E(\vec{f},g)$$

$$\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f},g),\, g = G(\vec{f},g,\vec{h}),\, \vec{h} = \vec{H}(\vec{f},g,\vec{h}) \textbf{ in } E(\vec{f},g)$$

Let-lifting rules

Let-lifting HRS $\mathbf{R}_{\text{let}\nearrow}$ with rewrite relation $\xrightarrow{}_{\text{let}\nearrow}$:

$$(_{\text{let}}\nearrow @_0) \quad (\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f}))\, E_1 \;\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f})\, E_1$$

$$(_{\text{let}}\nearrow @_1) \quad E_0\, (\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_1(\vec{f})) \;\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0\, E_1(\vec{f})$$

$$(_{\text{let}}\nearrow \lambda) \quad \lambda x.\, \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f},\vec{g},x) \textbf{ in } E(\vec{f},\vec{g},x)$$

$$\to \begin{cases} \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, E(\vec{f},x) & \text{if } \vec{g} \text{ is empty} \\ \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, \textbf{let } \vec{g} = \vec{G}(\vec{f},\vec{g},x) \textbf{ in } E(\vec{f},\vec{g},x) \end{cases}$$

$$(\textbf{let-in}_{-\,\text{let}}\nearrow) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in let } \vec{g} = \vec{G}(\vec{f},\vec{g}) \textbf{ in } E(\vec{f},\vec{g})$$

$$\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f},\vec{g}) \textbf{ in } E(\vec{f},\vec{g})$$

$$(\textbf{let}_{-\,\text{let}}\nearrow) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f},g),\, g = \textbf{let } \vec{h} = \vec{H}(\vec{f},g,\vec{h}) \textbf{ in } G(\vec{f},g,\vec{h}) \textbf{ in } E(\vec{f},g)$$

$$\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f},g),\, g = G(\vec{f},g,\vec{h}),\, \vec{h} = \vec{H}(\vec{f},g,\vec{h}) \textbf{ in } E(\vec{f},g)$$

Let-lifting rules

Let-lifting HRS $\mathbf{R}_{\text{let}\nearrow}$ with rewrite relation $\rightarrow_{\text{let}\nearrow}$:

$$(_{\text{let}}\nearrow @_0) \quad (\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f}))\, E_1 \;\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f})\, E_1$$

$$(_{\text{let}}\nearrow @_1) \quad E_0\,(\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_1(\vec{f})) \;\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0\, E_1(\vec{f})$$

$$(_{\text{let}}\nearrow \lambda) \quad \lambda x.\, \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f},\vec{g},x) \textbf{ in } E(\vec{f},\vec{g},x)$$

$$\to \begin{cases} \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, E(\vec{f},x) & \text{if } \vec{g} \text{ is empty} \\ \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, \textbf{let } \vec{g} = \vec{G}(\vec{f},\vec{g},x) \textbf{ in } E(\vec{f},\vec{g},x) \end{cases}$$

$$(\textbf{let-in}_{-\,\text{let}}\nearrow) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in let } \vec{g} = \vec{G}(\vec{f},\vec{g}) \textbf{ in } E(\vec{f},\vec{g})$$

$$\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f},\vec{g}) \textbf{ in } E(\vec{f},\vec{g})$$

$$(\textbf{let}_{-\,\text{let}}\nearrow) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f},g),\, g = \textbf{let } \vec{h} = \vec{H}(\vec{f},g,\vec{h}) \textbf{ in } G(\vec{f},g,\vec{h}) \textbf{ in } E(\vec{f},g)$$

$$\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f},g),\, g = G(\vec{f},g,\vec{h}),\, \vec{h} = \vec{H}(\vec{f},g,\vec{h}) \textbf{ in } E(\vec{f},g)$$

Let-lifting rules

Let-lifting HRS $\mathbf{R}_{\text{let}\nearrow}$ with rewrite relation $\rightarrow_{\text{let}\nearrow}$:

$$(_{\text{let}}\nearrow @_0) \quad (\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f}))\, E_1 \;\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f})\, E_1$$

$$(_{\text{let}}\nearrow @_1) \quad E_0\, (\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_1(\vec{f})) \;\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0\, E_1(\vec{f})$$

$$(_{\text{let}}\nearrow \lambda) \quad \lambda x.\, \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f}, \vec{g}, x) \textbf{ in } E(\vec{f}, \vec{g}, x)$$

$$\to \begin{cases} \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, E(\vec{f}, x) & \text{if } \vec{g} \text{ is empty} \\ \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, \textbf{let } \vec{g} = \vec{G}(\vec{f}, \vec{g}, x) \textbf{ in } E(\vec{f}, \vec{g}, x) & \end{cases}$$

$$(\textbf{let-in}_{-\,\text{let}}\nearrow) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in let } \vec{g} = \vec{G}(\vec{f}, \vec{g}) \textbf{ in } E(\vec{f}, \vec{g})$$

$$\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f}, \vec{g}) \textbf{ in } E(\vec{f}, \vec{g})$$

$$(\textbf{let}_{-\,\text{let}}\nearrow) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f}, g),\, g = \textbf{let } \vec{h} = \vec{H}(\vec{f}, g, \vec{h}) \textbf{ in } G(\vec{f}, g, \vec{h}) \textbf{ in } E(\vec{f}, g)$$

$$\to\; \textbf{let } \vec{f} = \vec{F}(\vec{f}, g),\, g = G(\vec{f}, g, \vec{h}),\, \vec{h} = \vec{H}(\vec{f}, g, \vec{h}) \textbf{ in } E(\vec{f}, g)$$

Let-lifting rules

Let-lifting HRS $\mathbf{R}_{\text{let}\nearrow}$ with rewrite relation $\rightarrow_{\text{let}\nearrow}$:

$$(\text{let}\nearrow @_0) \quad (\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f}))\, E_1 \;\rightarrow\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0(\vec{f})\, E_1$$

$$(\text{let}\nearrow @_1) \quad E_0\,(\textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_1(\vec{f})) \;\rightarrow\; \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } E_0\, E_1(\vec{f})$$

$$(\text{let}\nearrow \lambda) \quad \lambda x.\, \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f},\vec{g},x) \textbf{ in } E(\vec{f},\vec{g},x)$$

$$\rightarrow \begin{cases} \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, E(\vec{f},x) & \text{if } \vec{g} \text{ is empty} \\ \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in } \lambda x.\, \textbf{let } \vec{g} = \vec{G}(\vec{f},\vec{g},x) \textbf{ in } E(\vec{f},\vec{g},x) \end{cases}$$

$$(\textbf{let-in}_{\text{-}\ \text{let}\nearrow}) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f}) \textbf{ in let } \vec{g} = \vec{G}(\vec{f},\vec{g}) \textbf{ in } E(\vec{f},\vec{g})$$

$$\rightarrow\; \textbf{let } \vec{f} = \vec{F}(\vec{f}),\, \vec{g} = \vec{G}(\vec{f},\vec{g}) \textbf{ in } E(\vec{f},\vec{g})$$

$$(\textbf{let}_{\text{-}\ \text{let}\nearrow}) \quad \textbf{let } \vec{f} = \vec{F}(\vec{f},g),\, g = \textbf{let } \vec{h} = \vec{H}(\vec{f},g,\vec{h}) \textbf{ in } G(\vec{f},g,\vec{h}) \textbf{ in } E(\vec{f},g)$$

$$\rightarrow\; \textbf{let } \vec{f} = \vec{F}(\vec{f},g),\, g = G(\vec{f},g,\vec{h}),\, \vec{h} = \vec{H}(\vec{f},g,\vec{h}) \textbf{ in } E(\vec{f},g)$$

Let-lifting rewrite relations

Needed: conversion $=_{\mathsf{ex}}$ induced by rule:

$$\text{(exchange)} \quad \textbf{let } B_0,\ f_i = F_i(\vec{f}),\ f_{i+1} = F_{i+1}(\vec{f}),\ B_1 \textbf{ in } E(\vec{f})$$

$$\rightarrow\ \textbf{let } B_0,\ f_{i+1} = F_{i+1}(\vec{f}),\ f_i = F_i(\vec{f}),\ B_1 \textbf{ in } E(\vec{f})$$

Let-lifting rewrite relations

Needed: conversion $=_{\mathrm{ex}}$ induced by rule:

$$\text{(exchange)} \quad \mathbf{let}\ B_0,\ f_i = F_i(\vec{f}),\ f_{i+1} = F_{i+1}(\vec{f}),\ B_1\ \mathbf{in}\ E(\vec{f})$$
$$\to\ \mathbf{let}\ B_0,\ f_{i+1} = F_{i+1}(\vec{f}),\ f_i = F_i(\vec{f}),\ B_1\ \mathbf{in}\ E(\vec{f})$$

Define:

$$L \ {}_{\mathrm{let}}\!\nearrow\ L' \quad :\Longleftrightarrow \quad L \ =_{\mathrm{ex}} \cdot {}_{\mathrm{let}}\!\nearrow \cdot =_{\mathrm{ex}} L' \quad ({}_{\mathrm{let}}\!\nearrow \text{ modulo } =_{\mathrm{ex}})$$

Let-lifting rewrite relations

Needed: conversion $=_{\text{ex}}$ induced by rule:

$$(\text{exchange}) \quad \textbf{let } B_0,\ f_i = F_i(\vec{f}),\ f_{i+1} = F_{i+1}(\vec{f}),\ B_1 \textbf{ in } E(\vec{f})$$
$$\to\ \textbf{let } B_0,\ f_{i+1} = F_{i+1}(\vec{f}),\ f_i = F_i(\vec{f}),\ B_1 \textbf{ in } E(\vec{f})$$

Define:

$$L \ {}_{\text{let}}\!\nearrow\ L' \quad :\Longleftrightarrow \quad L \ =_{\text{ex}} \cdot {}_{\text{let}}\!\nearrow \cdot =_{\text{ex}} L' \qquad ({}_{\text{let}}\!\nearrow \text{ modulo } =_{\text{ex}})$$

$$[L]_{=_{\text{ex}}} \ {}_{[\text{let}]}\!\nearrow\ [L']_{=_{\text{ex}}} \quad :\Longleftrightarrow \quad L \ {}_{\text{let}}\!\nearrow\ L' \qquad (\text{on } =_{\text{ex}}\text{-equivalence classes})$$

Let-lifting rewrite relations

Needed: conversion $=_{\text{ex}}$ induced by rule:

$$(\text{exchange}) \quad \textbf{let } B_0,\ f_i = F_i(\vec{f}),\ f_{i+1} = F_{i+1}(\vec{f}),\ B_1 \textbf{ in } E(\vec{f}) \\ \to \ \textbf{let } B_0,\ f_{i+1} = F_{i+1}(\vec{f}),\ f_i = F_i(\vec{f}),\ B_1 \textbf{ in } E(\vec{f})$$

Define:

$$L \ {}_{\text{let}}\!\nearrow\ L' \quad :\Longleftrightarrow \quad L \ =_{\text{ex}} \cdot {}_{\text{let}}\!\nearrow \cdot =_{\text{ex}} L' \qquad ({}_{\text{let}}\!\nearrow \text{ modulo } =_{\text{ex}})$$

$$[L]_{=_{\text{ex}}} \ {}_{[\text{let}]}\!\nearrow\ [L']_{=_{\text{ex}}} \quad :\Longleftrightarrow \quad L \ {}_{\text{let}}\!\nearrow\ L' \qquad (\text{on } =_{\text{ex}}\text{-equivalence classes})$$

$\to$ is called locally confluent modulo $\sim$ if $\leftarrow \cdot \to \ \subseteq\ \twoheadrightarrow \cdot \sim \cdot \twoheadleftarrow$.

Lemma

(i) ${}_{\text{let}}\!\nearrow$ *is locally confluent modulo* $=_{\text{ex}}$.

(ii) ${}_{[\text{let}]}\!\nearrow$ *is locally confluent.*

Critical pair example

Proof.

(i) define HRS $\mathbf{R}_{\text{let}\nearrow_{\text{ex}}}$ with rewrite rel. $=_{\text{ex}} \hookrightarrow_{\text{let}\nearrow}$ [Peterson, Stickel,'81]

- rule scheme (σ) of $\mathbf{R}_{\text{let}\nearrow}$ $\longmapsto$ rule scheme $(\sigma)_{=_{\text{ex}}}$ of $\mathbf{R}_{\text{let}\nearrow_{\text{ex}}}$

Critical pair example

Proof.

(i) define HRS $\mathbf{R}_{\mathsf{let}\nearrow_{\mathsf{ex}}}$ with rewrite rel. ${}_{=_{\mathsf{ex}}}\hookrightarrow_{\mathsf{let}\nearrow}$ [Peterson, Stickel,'81]

- rule scheme (σ) of $\mathbf{R}_{\mathsf{let}\nearrow}$ $\longmapsto$ rule scheme $(\sigma)_{=_{\mathsf{ex}}}$ of $\mathbf{R}_{\mathsf{let}\nearrow_{\mathsf{ex}}}$

(ii) carry out a critical pair analysis

Critical pair example

Proof.

(i) define HRS $\mathbf{R}_{\text{let}\nearrow\text{ex}}$ with rewrite rel. ${=_{\text{ex}}} \hookrightarrow_{\text{let}\nearrow}$ [Peterson, Stickel,'81]

- rule scheme (σ) of $\mathbf{R}_{\text{let}\nearrow}$ $\longmapsto$ rule scheme $(\sigma)_{=_{\text{ex}}}$ of $\mathbf{R}_{\text{let}\nearrow\text{ex}}$

(ii) carry out a critical pair analysis

$(\text{let}\nearrow @_0)_{=_{\text{ex}}}$ / $(\text{let}\nearrow @_1)_{=_{\text{ex}}}$:

$$
\begin{array}{ccc}
(\textbf{let } \vec{f} = F(\vec{f}) \textbf{ in } E_0(\vec{f}))\,(\textbf{let } \vec{g} = G(\vec{g}) \textbf{ in } E_1(\vec{g})) & \xrightarrow[(\text{let}\nearrow @_0)]{} & \textbf{let } \vec{f} = F(\vec{f}) \textbf{ in } E_0(\vec{f})\,\textbf{let } \vec{g} = G(\vec{g}) \textbf{ in } E_1(\vec{g}) \\
\downarrow (\text{let}\nearrow @_1) & & \vdots\, (\text{let}\nearrow @_1) \\
\textbf{let } \vec{g} = G(\vec{g}) \textbf{ in } (\textbf{let } \vec{f} = F(\vec{f}) \textbf{ in } E_0(\vec{f}))\,E_1(\vec{g}) & & \textbf{let } \vec{f} = F(\vec{f}) \textbf{ in let } \vec{g} = G(\vec{g}) \textbf{ in } E_0(\vec{f})\,E_1(\vec{g}) \\
\vdots\, (\text{let}\nearrow @_0) & & \vdots\, (\textbf{let-in}_{-\ \text{let}\nearrow}) \cdot =_{\text{ex}} \\
\textbf{let } \vec{g} = G(\vec{g}) \textbf{ in let } \vec{f} = F(\vec{f}) \textbf{ in } E_0(\vec{f})\,E_1(\vec{g}) & \xrightarrow[(\textbf{let-in}_{-\ \text{let}\nearrow})]{} & \textbf{let } \vec{g} = G(\vec{g}),\ \vec{f} = F(\vec{f}) \textbf{ in } E_0(\vec{f})\,E_1(\vec{g})
\end{array}
$$

Critical pair example

Proof.

(i) define HRS $\mathbf{R}_{\mathsf{let}\nearrow_{\mathsf{ex}}}$ with rewrite rel. ${=_{\mathsf{ex}}}\hookrightarrow_{\mathsf{let}\nearrow}$ [Peterson, Stickel,'81]

- rule scheme (σ) of $\mathbf{R}_{\mathsf{let}\nearrow}$ $\longmapsto$ rule scheme $(\sigma)_{=_{\mathsf{ex}}}$ of $\mathbf{R}_{\mathsf{let}\nearrow_{\mathsf{ex}}}$

(ii) carry out a critical pair analysis

(iii) Critical Pair Theorem for HRS [Mayr, Nipkow,'96] implies local confluence of ${=_{\mathsf{ex}}}\hookrightarrow_{\mathsf{let}\nearrow}$

$(\mathsf{let}\nearrow @_0)_{=_{\mathsf{ex}}}$ / $(\mathsf{let}\nearrow @_1)_{=_{\mathsf{ex}}}$:

$$
\begin{array}{ccc}
(\mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f}))\,(\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_1(\vec{g})) & \xrightarrow[(\mathsf{let}\nearrow @_0)]{} & \mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\ \mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_1(\vec{g}) \\
\downarrow (\mathsf{let}\nearrow @_1) & & \downarrow (\mathsf{let}\nearrow @_1) \\
\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ (\mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f}))\ E_1(\vec{g}) & & \mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ \mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_0(\vec{f})\ E_1(\vec{g}) \\
\downarrow (\mathsf{let}\nearrow @_0) & & \downarrow (\mathbf{let\text{-}in}_{-\ \mathsf{let}\nearrow}) \cdot =_{\mathsf{ex}} \\
\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ \mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\ E_1(\vec{g}) & \xrightarrow[(\mathbf{let\text{-}in}_{-\ \mathsf{let}\nearrow})]{} & \mathbf{let}\ \vec{g} = G(\vec{g}),\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\ E_1(\vec{g})
\end{array}
$$

Critical pair example

Proof.

(i) define HRS $\mathbf{R}_{\mathrm{let}\nearrow\mathrm{ex}}$ with rewrite rel. $=_{\mathrm{ex}}\hookrightarrow_{\mathrm{let}\nearrow}$ [Peterson, Stickel,'81]

- rule scheme (σ) of $\mathbf{R}_{\mathrm{let}\nearrow}$ $\longmapsto$ rule scheme $(\sigma)_{=_{\mathrm{ex}}}$ of $\mathbf{R}_{\mathrm{let}\nearrow\mathrm{ex}}$

(ii) carry out a critical pair analysis

(iii) Critical Pair Theorem for HRS [Mayr, Nipkow,'96] implies local confluence of $=_{\mathrm{ex}}\hookrightarrow_{\mathrm{let}\nearrow}$

(iv) $_{\mathrm{let}\nearrow}$-steps and $=_{\mathrm{ex}}$-steps at different positions commute

(v) then it follows:
local confluence of $_{\mathrm{let}}\nearrow$ modulo $=_{\mathrm{ex}}$, and local confluence of $_{[\mathrm{let}]}\nearrow$

$(_{\mathrm{let}}\nearrow\ @_0)_{=_{\mathrm{ex}}}\ /\ (_{\mathrm{let}}\nearrow\ @_1)_{=_{\mathrm{ex}}}$:

$$(\mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f}))\,(\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_1(\vec{g})) \xrightarrow[(_{\mathrm{let}}\nearrow\ @_0)]{} \mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\,\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_1(\vec{g})$$

$$(\mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f}))\,(\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_1(\vec{g})) \xrightarrow{(_{\mathrm{let}}\nearrow\ @_1)} \mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ (\mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f}))\,E_1(\vec{g})$$

$$\mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\,\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_1(\vec{g}) \xrightarrow{(_{\mathrm{let}}\nearrow\ @_1)} \mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ \mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_0(\vec{f})\,E_1(\vec{g})$$

$$\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ (\mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f}))\,E_1(\vec{g}) \xrightarrow{(_{\mathrm{let}}\nearrow\ @_0)} \mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ \mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\,E_1(\vec{g})$$

$$\mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ \mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ E_0(\vec{f})\,E_1(\vec{g}) \xrightarrow{(\mathbf{let\text{-}in}_{-\ \mathrm{let}}\nearrow)\cdot =_{\mathrm{ex}}} \mathbf{let}\ \vec{g} = G(\vec{g}),\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\,E_1(\vec{g})$$

$$\mathbf{let}\ \vec{g} = G(\vec{g})\ \mathbf{in}\ \mathbf{let}\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\,E_1(\vec{g}) \xrightarrow[(\mathbf{let\text{-}in}_{-\ \mathrm{let}}\nearrow)]{} \mathbf{let}\ \vec{g} = G(\vec{g}),\ \vec{f} = F(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\,E_1(\vec{g})$$

Let-lifting is confluent

Lemma

$_{\text{let}}\!\nearrow$ *and* $_{[\text{let}]}\!\nearrow$ *are terminating.*

Proposition

In every $_{\text{let}}\!\nearrow$ or $_{[\text{let}]}\!\nearrow$-normal form, **let**-subterms occur only:

- at the root;
- immediately below λ-abstractions.

Theorem

$_{[\text{let}]}\!\nearrow$ *is confluent, terminating, and uniquely normalizing.*

Proof.

By using Newman's Lemma. □

Let-sinking

Applications may 'block' $\overset{\text{let}}{\searrow}$-steps, but not abstractions:

$$\textbf{let } f = \lambda y.\, y \textbf{ in } \lambda x.\, f\, f\, x$$

Let-sinking

Applications may 'block' $\overset{\text{let}}{\searrow}$-steps, but not abstractions:

$$\textbf{let } f = \lambda y.\, y \textbf{ in } \lambda x.\, f\, f\, x$$

$$\overset{\text{let}}{\searrow}\ \lambda x.\, \textbf{let } f = \lambda y.\, y \textbf{ in } f\, f\, x \qquad \text{(let-sinking over abstraction)}$$

Let-sinking

Applications may 'block' $\overset{\text{let}}{\searrow}$-steps, but not abstractions:

$$\textbf{let } f = \lambda y.\, y \textbf{ in } \lambda x.\, f\ f\ x$$

$$\overset{\text{let}}{\searrow}\ \lambda x.\, \textbf{let } f = \lambda y.\, y \textbf{ in } f\ f\ x \qquad \text{(let-sinking over abstraction)}$$

$$\overset{\text{let}}{\searrow}\ \lambda x.\, (\textbf{let } f = \lambda y.\, y \textbf{ in } f\ f\ )x \qquad \text{(let-sinking over application)}$$

Let-sinking

Applications may 'block' $\overset{\text{let}}{\searrow}$-steps, but not abstractions:

$$\textbf{let } f = \lambda y.\, y \textbf{ in } \lambda x.\, f\, f\, x$$

$$\overset{\text{let}}{\searrow} \ \lambda x.\, \textbf{let } f = \lambda y.\, y \textbf{ in } f\, f\, x \qquad \text{(let-sinking over abstraction)}$$

$$\overset{\text{let}}{\searrow} \ \lambda x.\, (\textbf{let } f = \lambda y.\, y \textbf{ in } f\, f\,)\, x \qquad \text{(let-sinking over application)}$$

in the sense that further sinking needs duplication:

$$\lambda x.\, (\textbf{let } f = \lambda y.\, y \textbf{ in } f)\, (\textbf{let } f = \lambda y.\, y \textbf{ in } f)\, x \qquad \text{(unfolding)}$$

which decreases (here looses) sharing (changes graph interpretation).

Let-sinking rules

Let-sinking HRS $\mathbf{R}^{\mathrm{let}\searrow}$ with rewrite relation $\to_{\mathrm{let}\searrow}$:

$$(_{\mathrm{let}}\nearrow @_0)\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E_0(\vec{f},\vec{g})\ E_1(\vec{f})$$

$$\to \begin{cases} \left(\mathbf{let}\ \vec{g} = \vec{G}(\vec{g})\ \mathbf{in}\ E_0(\vec{g})\right) E_1 & \text{if } \vec{f} \text{ is empty} \\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \left(\mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E_0(\vec{f},\vec{g})\right) E_1(\vec{f}) & \end{cases}$$

$$(_{\mathrm{let}}\nearrow @_1)\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E_0(\vec{f})\ E_1(\vec{f},\vec{g})$$

$$\to \begin{cases} E_0\left(\mathbf{let}\ \vec{g} = \vec{G}(\vec{g})\ \mathbf{in}\ E_1(\vec{g})\right) & \text{if } \vec{f} \text{ is empty} \\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E_0(\vec{f})\left(\mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E_1(\vec{f},\vec{g})\right) & \end{cases}$$

$$(^{\mathrm{let}}\searrow \lambda)\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \lambda x.\, E(\vec{f},x) \ \to\ \lambda x.\, \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E(\vec{f},x)$$

$$(^{\mathrm{let}}\searrow \mathbf{let}_{\_})\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ \mathbf{let}\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E(\vec{f},\vec{g})$$

$$\to\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E(\vec{f},\vec{g})$$

$$(\mathbf{let}_{\_}\,^{\mathrm{let}}\searrow)\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = G(\vec{f},g,\vec{h}),\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \mathbf{in}\ E(\vec{f},g)$$

$$\to\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f},g),\ g = \mathbf{let}\ \vec{h} = \vec{H}(\vec{f},g,\vec{h})\ \mathbf{in}\ G(\vec{f},g,\vec{h})\ \mathbf{in}\ E(\vec{f},g)$$

Garbage collection

$$\lambda x.\, \lambda y.\, \textbf{let}\ f = \lambda z.\, z\ \textbf{in}\ x\ y$$

Garbage collection

$$\lambda x.\, \lambda y.\, \textbf{let}\ f = \lambda z.\, z\ \textbf{in}\ x\, y$$

$$\swarrow^{\text{let}} \qquad\qquad\qquad\qquad {}^{\text{let}}\searrow$$

$$\lambda x.\, \lambda y.\, (\textbf{let}\ f = \lambda z.\, z\ \textbf{in}\ x)\, y \qquad\qquad \lambda x.\, \lambda y.\, x\, (\textbf{let}\ f = \lambda z.\, z\ \textbf{in}\ y)$$

Garbage collection

$$\lambda x.\, \lambda y.\, \textbf{let}\ f = \lambda z.\, z\ \textbf{in}\ x\ y$$

$$\swarrow^{\text{let}} \qquad\qquad\qquad\qquad {}^{\text{let}}\searrow$$

$$\lambda x.\, \lambda y.\, (\textbf{let}\ f = \lambda z.\, z\ \textbf{in}\ x)\ y \qquad\qquad \lambda x.\, \lambda y.\, x\, (\textbf{let}\ f = \lambda z.\, z\ \textbf{in}\ y)$$

Needed: garbage collection rules with rewrite relation $\to_{\text{gc}}$

$$(\text{reduce}) \quad \textbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f}, \vec{g})\ \textbf{in}\ E(\vec{f}) \ \to\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ E(\vec{f})$$

Garbage collection

$$\lambda x.\,\lambda y.\,\mathbf{let}\ f = \lambda z.\,z\ \mathbf{in}\ x\,y$$

$$\swarrow^{\text{let}} \qquad\qquad\qquad\qquad {}^{\text{let}}\searrow$$

$$\lambda x.\,\lambda y.\,(\mathbf{let}\ f = \lambda z.\,z\ \mathbf{in}\ x)\,y \qquad\qquad \lambda x.\,\lambda y.\,x\,(\mathbf{let}\ f = \lambda z.\,z\ \mathbf{in}\ y)$$

$$\to_{\text{gc}} \qquad\qquad\qquad\qquad \leftarrow_{\text{gc}}$$

$$\lambda x.\,\lambda y.\,(\mathbf{let}\ \ \mathbf{in}\ x)\,y \qquad \lambda x.\,\lambda y.\,x\,(\mathbf{let}\ \ \mathbf{in}\ y)$$

Needed: garbage collection rules with rewrite relation $\to_{\text{gc}}$

$$\text{(reduce)}\quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E(\vec{f}) \ \to\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E(\vec{f})$$

Garbage collection

$$\lambda x.\,\lambda y.\,\mathbf{let}\ f = \lambda z.\,z\ \mathbf{in}\ x\,y$$

$$\swarrow^{\text{let}} \qquad\qquad\qquad\qquad {}^{\text{let}}\searrow$$

$$\lambda x.\,\lambda y.\,(\mathbf{let}\ f = \lambda z.\,z\ \mathbf{in}\ x)\,y \qquad\qquad \lambda x.\,\lambda y.\,x\,(\mathbf{let}\ f = \lambda z.\,z\ \mathbf{in}\ y)$$

$$\to_{\text{gc}} \qquad\qquad\qquad\qquad \leftarrow_{\text{gc}}$$

$$\lambda x.\,\lambda y.\,(\mathbf{let}\ \ \mathbf{in}\ x)\,y \qquad\qquad \lambda x.\,\lambda y.\,x\,(\mathbf{let}\ \ \mathbf{in}\ y)$$

Needed: garbage collection rules with rewrite relation $\to_{\text{gc}}$

$$(\text{reduce}) \quad \mathbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \mathbf{in}\ E(\vec{f}) \ \to\ \mathbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \mathbf{in}\ E(\vec{f})$$

$$(\text{nil}) \quad \mathbf{let}\ \ \mathbf{in}\ L \ \to\ L$$

Garbage collection

$$\lambda x.\,\lambda y.\,\textbf{let}\ f = \lambda z.\,z\ \textbf{in}\ x\,y$$

$\swarrow^{\text{let}}$ $\quad\quad$ ${}^{\text{let}}\searrow$

$$\lambda x.\,\lambda y.\,(\textbf{let}\ f = \lambda z.\,z\ \textbf{in}\ x)\,y \qquad \lambda x.\,\lambda y.\,x\,(\textbf{let}\ f = \lambda z.\,z\ \textbf{in}\ y)$$

$\rightarrow_{\text{gc}}$ $\quad\quad$ $\leftarrow_{\text{gc}}$

$$\lambda x.\,\lambda y.\,(\textbf{let}\ \ \textbf{in}\ x)\,y \qquad \lambda x.\,\lambda y.\,x\,(\textbf{let}\ \ \textbf{in}\ y)$$

$\rightarrow_{\text{gc}}$ $\quad\quad$ $\leftarrow_{\text{gc}}$

$$\lambda x.\,\lambda y.\,x\,y$$

Needed: garbage collection rules with rewrite relation $\rightarrow_{\text{gc}}$

$$\text{(reduce)}\quad \textbf{let}\ \vec{f} = \vec{F}(\vec{f}),\ \vec{g} = \vec{G}(\vec{f},\vec{g})\ \textbf{in}\ E(\vec{f}) \ \rightarrow\ \textbf{let}\ \vec{f} = \vec{F}(\vec{f})\ \textbf{in}\ E(\vec{f})$$

$$\text{(nil)}\quad \textbf{let}\ \ \textbf{in}\ L \ \rightarrow\ L$$

Let-sinking is confluent

$$L \;{}^{\text{let}}\!\searrow^{\text{gc}}\; L' \quad :\Longleftrightarrow \quad L =_{\text{ex}} \cdot ({}_{\text{let}}\!\searrow \cup \to_{\text{gc}}) \cdot =_{\text{ex}} L' \qquad (({}_{\text{let}}\!\searrow \cup \to_{\text{gc}}) \text{ modulo } =_{\text{ex}})$$

$$[L]_{=_{\text{ex}}} \;{}^{[\text{let}]}\!\searrow^{[\text{gc}]}\; [L']_{=_{\text{ex}}} \quad :\Longleftrightarrow \quad L \;{}^{\text{let}}\!\searrow^{\text{gc}}\; L' \qquad (\text{on } =_{\text{ex}}\text{-equivalence classes})$$

Lemma

${}^{\text{let}}\!\searrow^{\text{gc}}$ *is locally confluent modulo* $=_{\text{ex}}$, *and* ${}^{[\text{let}]}\!\searrow^{[\text{gc}]}$ *is locally confluent.*

Proposition

${}^{\text{let}}\!\searrow^{\text{gc}}$ and ${}^{[\text{let}]}\!\searrow^{[\text{gc}]}$ are terminating.

Theorem

${}^{[\text{let}]}\!\searrow^{[\text{gc}]}$ *is confluent, terminating, and uniquely normalizing.*

Envisaged application: lambda-lifting

Extend $\mathbf{R}_{\mathsf{let}\nearrow}$ with a parameter-addition rule:

$$
\begin{aligned}
&\lambda x.\,\mathbf{let}\ f = F(f, \vec{g}, x), \vec{g} = \vec{G}(f, \vec{g}, x)\ \mathbf{in}\ E(f, \vec{g}, x) \\
&\qquad \to\ \lambda x.\,\mathbf{let}\ \hat{f} = \lambda x'.\,F(\hat{f}\,x', \vec{g}, x'), \vec{g} = \vec{G}(\hat{f}\,x, \vec{g}, x)\ \mathbf{in}\ E(\hat{f}\,x, \vec{g}, x)
\end{aligned}
$$

to enable further let-lifting.

Envisaged application: lambda-lifting

Extend $\mathbf{R}_{\mathsf{let}\nearrow}$ with a parameter-addition rule:

$$\lambda x.\,\mathbf{let}\ f = F(f, \vec{g}, x), \vec{g} = \vec{G}(f, \vec{g}, x)\ \mathbf{in}\ E(f, \vec{g}, x)$$
$$\rightarrow\ \lambda x.\,\mathbf{let}\ \hat{f} = \lambda x'.\,F(\hat{f}\,x', \vec{g}, x'), \vec{g} = \vec{G}(\hat{f}\,x, \vec{g}, x)\ \mathbf{in}\ E(\hat{f}\,x, \vec{g}, x)$$

to enable further let-lifting.

Aim:

- enable to let-lift ('float out') all let-bindings
  to create a single outermost let-binding
- model a lambda-lifting translation into supercombinators
- show confluence modulo order of combinator arguments
- perhaps use normalized rewriting on let-floating equivalence classes

Summary

1. Let-lifting
   - let-lifting HRS $\mathbf{R}_{\text{let}\nearrow}$ with rewrite relation $_{\text{let}}\!\nearrow$
   - exchange conversion $=_{\text{ex}}$
   - rewrite relation $_{\text{let}}\!\nearrow\ := (=_{\text{ex}} \cdot {}_{\text{let}}\!\nearrow \cdot =_{\text{ex}})$ is confluent modulo $=_{\text{ex}}$
   - $=_{\text{ex}}$-class rewrite relation $_{[\text{let}]}\!\nearrow$ is confluent and terminating
2. Let-sinking rewrite relation $^{[\text{let}]}\!\searrow^{[\text{gc}]}$
   - let-sinking HRS $\mathbf{R}^{\text{let}\searrow}$ with rewrite relation $_{\text{let}}\!\searrow$
   - rewrite relation $^{\text{let}}\!\searrow^{\text{gc}} := =_{\text{ex}} \cdot ({}_{\text{let}}\!\nearrow \cup \rightarrow_{\text{gc}}) \cdot =_{\text{ex}}$ is confluent modulo $=_{\text{ex}}$
   - $=_{\text{ex}}$-class rewrite relation $^{[\text{let}]}\!\searrow^{[\text{gc}]}$ and confluent and terminating