Lazy evaluation

Péter Diviánszky

November 03, 2016

Introduction

Motivations

- to understand lazy evaluation
  - to understand purely functional language implementations (e.g. STG behind GHC)
- to be able to implement lazy evaluation efficiently

Lazy evaluation

Lazy evaluation is an *evaluation strategy* for λ-expressions.

Key feature:

- no work duplication
  - Caveat: work may be duplicated within λ-abstractions

Natural semantics of lazy evaluation

λ-calculus with mutually recursive let bindings

$$
\begin{array}{llll}
\mathcal{E} \ni e & = & v & \text{— variable} \\
 & \mid & e\ e' & \text{— application} \\
 & \mid & \lambda v \mapsto e & \text{— } \lambda\text{-abstraction} \\
 & \mid & \text{let}\ \Gamma\ \text{in}\ e & \text{— let expression} \\
\mathcal{B} \ni \Gamma, \Delta, \Theta & = & v_1 = e_1; \ldots; v_n = e_n & \text{— bindings}
\end{array}
$$

Examples:

- $\lambda x \mapsto x$
- $\lambda f\ x \mapsto f\ (f\ x) \quad \equiv \quad \lambda f \mapsto \lambda x \mapsto f\ (f\ x)$
- $\lambda f \mapsto \text{let}\ x = f\ x\ \text{in}\ x$
- $\text{let}\ x = x\ \text{in}\ x$
- $\text{let}\ x = y; y = x\ \text{in}\ x$

Variable representation

Possibilities:

1. $v \in \mathcal{V}$, an infinite set of symbols
   - renaming of variables is needed to avoid name conflicts
2. each variable is the same and we count shadowings (called De-Bruijn indices)
   - recalculation of shadowings is needed in operations

Variable representation is an orthogonal choice which we postpone.

More extensions to λ-calculus

$$
\begin{array}{rcll}
\mathcal{E} \ni e & = & \dots & \\
& | & \dots \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots & \text{— integer literals} \\
& | & e + e' & \text{— addition} \\
& | & e * e' & \text{— multiplication}
\end{array}
$$

Examples:

- $1 + 2 * 3$
- $\text{let}\, x = 1 + 1 \,\text{in}\, x + x$

Evaluation judgement

$$e \Downarrow e'$$

Examples:

- $1 \Downarrow 1$
- $1 + 2 * 3 \Downarrow 7$
- $\text{let}\, x = 1 + 1 \,\text{in}\, x + x \Downarrow 4$
- $\lambda x \mapsto x \Downarrow \lambda x \mapsto x$
- $\lambda x \mapsto 1 + 1 \Downarrow \lambda x \mapsto 1 + 1$
- $(\lambda x \mapsto x)\,(\lambda y \mapsto y) \Downarrow \lambda y \mapsto y$
- $\text{let}\, x = 1 + 1 \,\text{in}\, x + z \Downarrow 2 + z$
- $\text{let}\, x = 1 + 1 \,\text{in}\, z + x \Downarrow \text{let}\, x = 1 + 1 \,\text{in}\, z + x$

Heap

Let us call *heap* the top-level let bindings of expressions:

$$\text{let } \underbrace{x = 1; y = 2}_{\text{heap}} \text{ in } \underbrace{x + y}_{\text{main expression}}$$

We ensure that each expression has a heap:

$$e \quad \rightsquigarrow \quad \text{let } \mathit{main} = e \text{ in } \mathit{main}$$

or

$$e \quad \rightsquigarrow \quad \text{let } \varepsilon \text{ in } e$$

LAM-rule

λ-abstractions and literals are ready to use, nothing to do with them:

$$\frac{}{\mathsf{let}\,\Gamma\,\mathsf{in}\,n \;\Downarrow\; \mathsf{let}\,\Gamma\,\mathsf{in}\,n}\;\textsc{Lam}$$

n stands for a λ-abstraction or a literal.

Example:

$$\frac{}{\mathsf{let}\,\varepsilon\,\mathsf{in}\,\lambda y \mapsto y \;\Downarrow\; \mathsf{let}\,\varepsilon\,\mathsf{in}\,\lambda y \mapsto y}\;\textsc{Lam}$$

Let-rule

Let bidings are added to the heap:

$$\frac{\mathsf{let}\ \Gamma; \Delta\ \mathsf{in}\ e \ \Downarrow\ \mathsf{let}\ \Theta\ \mathsf{in}\ e'}{\mathsf{let}\ \Gamma\ \mathsf{in}\ (\mathsf{let}\ \Delta\ \mathsf{in}\ e) \ \Downarrow\ \mathsf{let}\ \Theta\ \mathsf{in}\ e'}\ \textsc{Let}$$

Example:

$$\frac{\dfrac{\dots}{\mathsf{let}\ x = 1 + 1; y = 1 * 1\ \mathsf{in}\ x + y \ \Downarrow\ \mathsf{let}\ x = 2; y = 1\ \mathsf{in}\ 3}}{\mathsf{let}\ x = 1 + 1\ \mathsf{in}\ (\mathsf{let}\ y = 1 * 1\ \mathsf{in}\ x + y) \ \Downarrow\ \mathsf{let}\ x = 2; y = 1\ \mathsf{in}\ 3}\ \textsc{Let}$$

Var-rule

Lookup the variable on the heap and evaluate it:

$$\frac{\text{let } \Gamma \text{ in } e \;\Downarrow\; \text{let } \Theta \text{ in } n}{\text{let } \Gamma; v = e \text{ in } v \;\Downarrow\; \text{let } \Theta; v = n \text{ in } n}\ \textsc{Var}$$

Example:

$$\frac{\dfrac{\cdots}{\text{let } \varepsilon \text{ in } 1+1 \;\Downarrow\; \text{let } \varepsilon \text{ in } 2}}{\text{let } x = 1+1 \text{ in } x \;\Downarrow\; \text{let } x = 2 \text{ in } 2}\ \textsc{Var}$$

$\textsc{App}$-rule

First evaluate the function, then the argument:

$$\frac{\text{let}\ \Gamma\ \text{in}\ e\ \Downarrow\ \text{let}\ \Delta\ \text{in}\ \lambda v \mapsto e'' \qquad \text{let}\ \Delta; v = e'\ \text{in}\ e''\ \Downarrow\ \text{let}\ \Theta\ \text{in}\ e'''}{\text{let}\ \Gamma\ \text{in}\ e\ e'\ \Downarrow\ \text{let}\ \Theta\ \text{in}\ e'''}\ \textsc{App}$$

Example:

$$\frac{\dfrac{\dots}{\text{let}\ f = \lambda y \mapsto y\ \text{in}\ f\ \Downarrow\ \text{let}\ f = \lambda y \mapsto y\ \text{in}\ \lambda y \mapsto y} \qquad \dfrac{\dots}{\text{let}\ f = \lambda y \mapsto y; y = 1\ \text{in}\ y\ \Downarrow\ \text{let}\ f = \lambda y \mapsto y; y = 1\ \text{in}\ 1}}{\text{let}\ f = \lambda y \mapsto y\ \text{in}\ f\ 1\ \Downarrow\ \text{let}\ f = \lambda y \mapsto y; y = 1\ \text{in}\ 1}\ \textsc{App}$$

Natural semantics of lazy evaluation (summary)

$$\frac{}{\mathsf{let}\ \Gamma\ \mathsf{in}\ n \Downarrow \mathsf{let}\ \Gamma\ \mathsf{in}\ n}\ \textsc{Lam}$$

$$\frac{\mathsf{let}\ \Gamma; \Delta\ \mathsf{in}\ e \Downarrow \mathsf{let}\ \Theta\ \mathsf{in}\ e'}{\mathsf{let}\ \Gamma\ \mathsf{in}\ (\mathsf{let}\ \Delta\ \mathsf{in}\ e) \Downarrow \mathsf{let}\ \Theta\ \mathsf{in}\ e'}\ \textsc{Let}$$

$$\frac{\mathsf{let}\ \Gamma\ \mathsf{in}\ e \Downarrow \mathsf{let}\ \Theta\ \mathsf{in}\ n}{\mathsf{let}\ \Gamma; v = e\ \mathsf{in}\ v \Downarrow \mathsf{let}\ \Theta; v = n\ \mathsf{in}\ n}\ \textsc{Var}$$

$$\frac{\mathsf{let}\ \Gamma\ \mathsf{in}\ e \Downarrow \mathsf{let}\ \Delta\ \mathsf{in}\ \lambda v \mapsto e'' \qquad \mathsf{let}\ \Delta; v = e'\ \mathsf{in}\ e'' \Downarrow \mathsf{let}\ \Theta\ \mathsf{in}\ e'''}{\mathsf{let}\ \Gamma\ \mathsf{in}\ e\ e' \Downarrow \mathsf{let}\ \Theta\ \mathsf{in}\ e'''}\ \textsc{App}$$

Extensions to natural semantics

Addition & multiplication

$$\frac{\mathsf{let}\ \Gamma\ \mathsf{in}\ e\ \Downarrow\ \mathsf{let}\ \Delta\ \mathsf{in}\ \ell \qquad \mathsf{let}\ \Delta\ \mathsf{in}\ e'\ \Downarrow\ \mathsf{let}\ \Theta\ \mathsf{in}\ \ell' \qquad \ell + \ell' = \ell''}{\mathsf{let}\ \Gamma\ \mathsf{in}\ e + e'\ \Downarrow\ \mathsf{let}\ \Theta\ \mathsf{in}\ \ell''}\ \textsc{Add}$$

$$\frac{\mathsf{let}\ \Gamma\ \mathsf{in}\ e\ \Downarrow\ \mathsf{let}\ \Delta\ \mathsf{in}\ \ell \qquad \mathsf{let}\ \Delta\ \mathsf{in}\ e'\ \Downarrow\ \mathsf{let}\ \Theta\ \mathsf{in}\ \ell' \qquad \ell * \ell' = \ell''}{\mathsf{let}\ \Gamma\ \mathsf{in}\ e * e'\ \Downarrow\ \mathsf{let}\ \Theta\ \mathsf{in}\ \ell''}\ \textsc{Mul}$$

Example:

$$\frac{\dfrac{\dfrac{\mathsf{let}\ \varepsilon\ \mathsf{in}\ 1\ \Downarrow\ \mathsf{let}\ \varepsilon\ \mathsf{in}\ 1 \qquad \mathsf{let}\ \varepsilon\ \mathsf{in}\ 2\ \Downarrow\ \mathsf{let}\ \varepsilon\ \mathsf{in}\ 2 \qquad 1 + 2 = 3}{\mathsf{let}\ \varepsilon\ \mathsf{in}\ 1 + 2\ \Downarrow\ \mathsf{let}\ \varepsilon\ \mathsf{in}\ 3}}{\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ x\ \Downarrow\ \mathsf{let}\ x = 3\ \mathsf{in}\ 3} \qquad \dfrac{\mathsf{let}\ \varepsilon\ \mathsf{in}\ 3\ \Downarrow\ \mathsf{let}\ \varepsilon\ \mathsf{in}\ 3}{\mathsf{let}\ x = 3\ \mathsf{in}\ x\ \Downarrow\ \mathsf{let}\ x = 3\ \mathsf{in}\ 3} \qquad 3 * 3 = 9}{\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ x * x\ \Downarrow\ \mathsf{let}\ x = 3\ \mathsf{in}\ 9}$$

Garbage collection

GC removes unused heap bindings:

$$\mathsf{let}\, {\color{red}x} = 1 \,\mathsf{in}\, 2 \quad \overset{\mathsf{GC}}{\rightsquigarrow} \quad \mathsf{let}\, \varepsilon \,\mathsf{in}\, 2$$

GC can be integrated with evaluation:

$$\mathsf{let}\, {\color{red}x} = 1 + 2 \,\mathsf{in}\, {\color{red}x} * {\color{red}x} \ \Downarrow_{\mathsf{GC}} \ \mathsf{let}\, \varepsilon \,\mathsf{in}\, 9$$

Intermediate GCs do not affect the result.

Cycle detection

$\mathcal{E} \ni e$	$=$	$\ldots$	
	$\mid$	$\perp$	— error

The $\textsc{Var}_\perp$-rule can handle cyclic evaluations:

$$\frac{\text{let}\,\Gamma;\, {\color{red}v} = \perp \text{in}\; e \;\Downarrow\; \text{let}\,\Theta;\, {\color{red}v} = \perp \text{in}\; n}{\text{let}\,\Gamma;\, {\color{red}v} = e \;\text{in}\; {\color{red}v} \;\Downarrow\; \text{let}\,\Theta;\, {\color{red}v} = n \;\text{in}\; n}\;\textsc{Var}_\perp$$

n stands for a λ-abstraction, a literal or an error.
Example:

$$\cfrac{\cfrac{\text{let}\; {\color{red}x} = \perp \text{in} \perp \;\Downarrow\; \text{let}\; {\color{red}x} = \perp \text{in} \perp}{\text{let}\; {\color{red}x} = \perp \text{in}\; {\color{red}x} \;\Downarrow\; \text{let}\; {\color{red}x} = \perp \text{in} \perp}\;\textsc{Var}_\perp}{\text{let}\; {\color{red}x} = {\color{red}x} \;\text{in}\; {\color{red}x} \;\Downarrow\; \text{let}\; {\color{red}x} = \perp \text{in} \perp}\;\textsc{Var}_\perp$$

Open expressions

Open expressions can be handled by adding more rules:

$$\frac{v \notin \mathrm{dom}\,\Gamma}{\mathsf{let}\,\Gamma\,\mathsf{in}\,v \;\Downarrow\; \mathsf{let}\,\Gamma\,\mathsf{in}\,v}\ \textsc{Var-open}$$

$$\frac{\mathsf{let}\,\Gamma;\, v = \bot\,\mathsf{in}\,e \;\Downarrow\; \mathsf{let}\,\Theta;\, v = \bot\,\mathsf{in}\,h}{\mathsf{let}\,\Gamma;\, v = e\,\mathsf{in}\,v \;\Downarrow\; \mathsf{let}\,\Theta;\, v = h\,\mathsf{in}\,v}\ \textsc{Var-h}$$

$$\frac{\mathsf{let}\,\Gamma\,\mathsf{in}\,e \;\Downarrow\; \mathsf{let}\,\Theta\,\mathsf{in}\,h}{\mathsf{let}\,\Gamma\,\mathsf{in}\,e\;\,e' \;\Downarrow\; \mathsf{let}\,\Theta\,\mathsf{in}\,h\;\,e'}\ \textsc{App-h}$$

h stands for an expression opposite of n, i.e. not λ-abstraction, literal or error.

Remark about work duplication

Only *n*-expressions are copied ever.

Copying a literal may not duplicate work.
Copying a λ-abstraction may duplicate work. For example, there will be two additions instead of one:

$$\text{let}\, f = \lambda x \mapsto 1 + 2 \,\text{in}\, f\, 0 * f\, 0 \;\Downarrow\; 9$$

Optimal reduction or full laziness optimization can help in this.

Deriving an abstract machine

Constructing derivation trees

For each expression, a derivation tree for $\Downarrow$ can be made mechanically.

Steps of construction

$$\frac{}{\underbrace{\text{let } \Gamma \text{ in } n}_{1} \Downarrow \underbrace{\text{let } \Gamma \text{ in } n}_{1}}\ \textsc{Lam}$$

$$\frac{\overbrace{\text{let } \Gamma ; \Delta \text{ in } e}^{2} \Downarrow \overbrace{\text{let } \Theta \text{ in } e'}^{3}}{\underbrace{\text{let } \Gamma \text{ in } (\text{let } \Delta \text{ in } e)}_{1} \Downarrow \underbrace{\text{let } \Theta \text{ in } e'}_{3}}\ \textsc{Let}$$

$$\frac{\overbrace{\text{let } \Gamma \text{ in } e}^{2} \Downarrow \overbrace{\text{let } \Theta \text{ in } n}^{3}}{\underbrace{\text{let } \Gamma ; v = e \text{ in } v}_{1} \Downarrow \underbrace{\text{let } \Theta ; v = n \text{ in } n}_{4}}\ \textsc{Var}$$

$$\frac{\overbrace{\text{let } \Gamma \text{ in } e}^{2} \Downarrow \overbrace{\text{let } \Delta \text{ in } \lambda v \mapsto e''}^{3} \qquad \overbrace{\text{let } \Delta ; v = e' \text{ in } e''}^{4} \Downarrow \overbrace{\text{let } \Theta \text{ in } e'''}^{5}}{\underbrace{\text{let } \Gamma \text{ in } e\ e'}_{1} \Downarrow \underbrace{\text{let } \Theta \text{ in } e'''}_{5}}\ \textsc{App}$$

Machine state

The current expression is not enough to continue the construction. The missing information in the rules:

$$\frac{\overbrace{\text{let } \Gamma \text{ in } e}^{\textcircled{2}\; v=\bullet} \Downarrow \overbrace{\text{let } \Theta \text{ in } n}^{\textcircled{3}\; v=\bullet}}{\underbrace{\text{let } \Gamma;\, v = e \text{ in } v}_{\textcircled{1}} \Downarrow \underbrace{\text{let } \Theta;\, v = n \text{ in } n}_{\textcircled{4}}}\ \textsc{Var}$$

$$\frac{\overbrace{\text{let } \Gamma \text{ in } e}^{\textcircled{2}\; \bullet\, e'} \Downarrow \overbrace{\text{let } \Delta \text{ in } \lambda v \mapsto e''}^{\textcircled{3}\; \bullet\, e'} \qquad \overbrace{\text{let } \Delta;\, v = e' \text{ in } e''}^{\textcircled{4}} \Downarrow \overbrace{\text{let } \Theta \text{ in } e'''}^{\textcircled{5}}}{\underbrace{\text{let } \Gamma \text{ in } e\ e'}_{\textcircled{1}} \Downarrow \underbrace{\text{let } \Theta \text{ in } e'''}_{\textcircled{5}}}\ \textsc{App}$$

We need a stack to store the missing information!

Expressions with stack

$$e; S$$

$$\frac{}{\text{let}\,\Gamma\,\text{in}\; n; S \;\Downarrow\; \text{let}\,\Gamma\,\text{in}\; n; S}\;\textsc{Lam}$$

$$\frac{\text{let}\,(\Gamma;\Delta)\,\text{in}\; e; S \;\Downarrow\; \text{let}\,\Theta\,\text{in}\; e'; S}{\text{let}\,\Gamma\,\text{in}\,(\text{let}\,\Delta\,\text{in}\; e); S \;\Downarrow\; \text{let}\,\Theta\,\text{in}\; e'; S}\;\textsc{Let}$$

$$\frac{\text{let}\,\Gamma\,\text{in}\; e; {\color{red}v} = \bullet, S \;\Downarrow\; \text{let}\,\Theta\,\text{in}\; n; {\color{red}v} = \bullet, S}{\text{let}\,(\Gamma; {\color{red}v} = e)\,\text{in}\; {\color{red}v}; S \;\Downarrow\; \text{let}\,(\Theta; {\color{red}v} = n)\,\text{in}\; n; S}\;\textsc{Var}$$

$$\frac{\text{let}\,\Gamma\,\text{in}\; e; \bullet\, e', S \;\Downarrow\; \text{let}\,\Delta\,\text{in}\; \lambda{\color{red}v} \mapsto e''; \bullet\, e', S \qquad \text{let}\,\Delta; {\color{red}v} = e'\,\text{in}\; e''; S \;\Downarrow\; \text{let}\,\Theta\,\text{in}\; e'''; S}{\text{let}\,\Gamma\,\text{in}\; e\;\, e'; S \;\Downarrow\; \text{let}\,\Theta\,\text{in}\; e'''; S}\;\textsc{App}$$

The abstract machine

Machine state

$$\text{let } \underbrace{\Gamma}_{\text{heap}} \text{ in } \underbrace{e}_{\text{target expression}} ; \underbrace{S}_{\text{stack}}$$

Machine rules

$\text{let } \Gamma \text{ in } (\text{let } \Delta \text{ in } e);\ S$	$\rightsquigarrow \text{let } (\Gamma; \Delta) \text{ in } e;\ S$	$\textsc{Let}$
$\text{let } (\Gamma; v = e) \text{ in } v;\ S$	$\rightsquigarrow \text{let } \Gamma \text{ in } e;\ v = \bullet, S$	$\textsc{Var}_1$
$\text{let } \Gamma \text{ in } n;\ v = \bullet, S$	$\rightsquigarrow \text{let } (\Gamma; v = n) \text{ in } n;\ S$	$\textsc{Var}_2$
$\text{let } \Gamma \text{ in } e\ e';\ S$	$\rightsquigarrow \text{let } \Gamma \text{ in } e;\ \bullet\, e', S$	$\textsc{App}_1$
$\text{let } \Gamma \text{ in } \lambda v \mapsto e'';\ \bullet\, e', S$	$\rightsquigarrow \text{let } (\Gamma; v = e') \text{ in } e'';\ S$	$\textsc{App}_2$

Machine rules handling addition

$\mathsf{let}\ \Gamma\ \mathsf{in}\ e + e';\ \ S$		$\rightsquigarrow$	$\mathsf{let}\ \Gamma\ \mathsf{in}\ e;\ \ \bullet + e', S$	$\textsc{Add}_1$
$\mathsf{let}\ \Gamma\ \mathsf{in}\ \ell;\ \ \bullet + e, S$		$\rightsquigarrow$	$\mathsf{let}\ \Gamma\ \mathsf{in}\ e;\ \ \ell + \bullet, S$	$\textsc{Add}_2$
$\mathsf{let}\ \Gamma\ \mathsf{in}\ \ell';\ \ \ell + \bullet, S$	$\ell + \ell' = \ell''$	$\rightsquigarrow$	$\mathsf{let}\ \Gamma\ \mathsf{in}\ \ell'';\ \ S$	$\textsc{Add}_3$

Example:

$\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ x * x;\quad \varepsilon$		$\textsc{Mul}_1 \rightsquigarrow$
$\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ x;\quad \bullet * x$		$\textsc{Var}_1 \rightsquigarrow$
$\mathsf{let}\ \varepsilon\ \mathsf{in}\ 1 + 2;\quad x = \bullet, \bullet * x$		$\textsc{Add}_1 \rightsquigarrow$
$\mathsf{let}\ \varepsilon\ \mathsf{in}\ 1;\quad \bullet + 2, x = \bullet, \bullet * x$		$\textsc{Add}_2 \rightsquigarrow$
$\mathsf{let}\ \varepsilon\ \mathsf{in}\ 2;\quad 1 + \bullet, x = \bullet, \bullet * x$	1+2=3	$\textsc{Add}_3 \rightsquigarrow$
$\mathsf{let}\ \varepsilon\ \mathsf{in}\ 3;\quad x = \bullet, \bullet * x$		$\textsc{Var}_2 \rightsquigarrow$
$\mathsf{let}\ x = 3\ \mathsf{in}\ 3;\quad \bullet * x$		$\textsc{Mul}_2 \rightsquigarrow$
$\mathsf{let}\ x = 3\ \mathsf{in}\ x;\quad 3 * \bullet$		$\textsc{Var}_1 \rightsquigarrow$
$\mathsf{let}\ \varepsilon\ \mathsf{in}\ 3;\quad x = \bullet, \bullet * x$		$\textsc{Var}_2 \rightsquigarrow$
$\mathsf{let}\ x = 3\ \mathsf{in}\ 3;\quad 3 * \bullet$	3*3=9	$\textsc{Mul}_3 \rightsquigarrow$
$\mathsf{let}\ x = 3\ \mathsf{in}\ 9;\quad \varepsilon$		

Reference

Peter Sestoft: *Deriving a Lazy Abstract Machine*
Journal of Functional Programming, 1997.

STG

STG overview

The Spineless Taggless G-machine is an abstract machine for lazy evaluation.

GHC compiler's code generation is based on STG.

Simon Peyton Jones: *Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine*
Journal of Functional Programming, 1992.

STG vs. the previous machine

STG has the following differences:

- update flag
- application on atoms only
- multi-arg application and λ-abstraction
- top-level λ-abstractions only

Update flag (optimization idea)

We infer by static analysis when no update on the heap is needed after evaluating a heap-expression.

Var-rule splits into two:

$$\frac{\text{let}\,\Gamma\,\text{in}\,e \;\Downarrow\; \text{let}\,\Theta\,\text{in}\,n}{\text{let}\,\Gamma;\, v_{\mathsf{u}} = e\,\text{in}\,v \;\Downarrow\; \text{let}\,\Theta;\, v = n\,\text{in}\,n}\ \textsc{Var-update}$$

$$\frac{\text{let}\,\Gamma\,\text{in}\,e \;\Downarrow\; \text{let}\,\Theta\,\text{in}\,n}{\text{let}\,\Gamma;\, v_{\mathsf{nu}} = e\,\text{in}\,v \;\Downarrow\; \text{let}\,\Theta\,\text{in}\,n}\ \textsc{Var-no-update}$$

Application on atoms only

$$
\begin{array}{llll}
\hline
\mathcal{E} \ni e & = & \dots & \\
& \mid & e\ a & \text{— application} \\
\mathcal{A} \ni a & = & v & \text{— variable} \\
& \mid & \ell & \text{— literal} \\
\hline
\end{array}
$$

Different $\textsc{App}$-rule:

$$
\frac{\text{let}\ \Gamma\ \text{in}\ e\ \Downarrow\ \text{let}\ \Delta\ \text{in}\ \lambda v \mapsto e' \qquad \text{let}\ \Delta\ \text{in}\ e'[v \mapsto a]\ \Downarrow\ \text{let}\ \Theta\ \text{in}\ e''}{\text{let}\ \Gamma\ \text{in}\ e\ a\ \Downarrow\ \text{let}\ \Theta\ \text{in}\ e''}\ \textsc{App'}
$$

- $e'[v \mapsto a]$ is implemented by *closures* to avoid substitution

Multi-arg application and λ-abstraction

$$
\begin{array}{rcll}
\mathcal{E} \ni e & = & \ldots & \\
 & | & {\color{red}v}\ a_1 \cdots a_n & \text{— application} \\
 & | & \lambda {\color{red}v_1 \cdots v_n} \mapsto e & \text{— } \lambda\text{-abstraction}
\end{array}
$$

More efficient, but needs more care (push/enter or eval/apply)

Top-level λ-abstractions only

λ-abstractions can not be inside STG expressions for efficiency and simplicity.

This is achieved by the λ-lifting transformation.

Missing from this presentation

Practical extensions to λ-calculus:

- more literals
- more builtin functions
  - foreign function interface
- constructors
  - algebraic data types
- case expression
  - pattern matching
- type annotation
  - typing

Questions