# Launchbury's Natural Semantics for Lazy Evaluation

Anthony M. Sloane

Programming Languages Research Group
Department of Computing, Macquarie University
Sydney, Australia

http://www.comp.mq.edu.au/~asloane
http://plrg.science.mq.edu.au

MACQUARIE UNIVERSITY | FACULTY OF SCIENCE

# A Natural Semantics for Lazy Evaluation

John Launchbury

In POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New York, NY, USA, 1993), ACM, pp. 144–154.

Aims to capture both non-strictness of evaluation and sharing of certain reductions.

Achieves a middle ground between more abstract semantic descriptions and the detailed operational semantics of abstract machines.

# Examples

$$let\ u = 3 + 2, v = u + 1\ in\ v + v$$

$$let\ u = 3 + 2, f = \lambda x.(let\ v = u + 1\ in\ v + x)\ in\ f\ 2 + f\ 3$$

$$let\ u = 3 + 2, f = (let\ v = u + 1\ in\ \lambda x.v + x)\ in\ f\ 2 + f\ 3$$

# Source language

Lambda calculus with recursive lets.

$$
\begin{array}{rcl}
x \in \mathit{Var} & & \\
e \in \mathit{Exp} & ::= & \lambda x.e \\
& | & e\ e \\
& | & x \\
& | & \mathit{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathit{in}\ e
\end{array}
$$

# Normalised language

Unique bound variable names and applications only to variables.

$$
\begin{array}{lcl}
x \in \mathit{Var} & & \\
e \in \mathit{Exp} & ::= & \lambda x.e \\
 & | & e\ x \\
 & | & x \\
 & | & \mathit{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathit{in}\ e
\end{array}
$$

$$
\begin{array}{lcl}
e\ x & \rightarrow & e\ x \\
e_1\ e_2 & \rightarrow & \mathit{let}\ y = e_2\ \mathit{in}\ e_1\ y \text{ (where } y \text{ is a fresh variable)}
\end{array}
$$

# Judgement

Naming conventions

$$\begin{array}{rclcl} \Gamma, \Delta, \Theta & \in & \mathit{Heap} & = & \mathit{Var} \rightarrow \mathit{Exp} \\ z & \in & \mathit{Val} & ::= & \lambda x.e \end{array}$$

Reduction

evaluating an expression in the context of a starting heap gives a value and a new heap

$$\Gamma : e \Downarrow \Delta : z$$

# Reduction rules

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e$$ (lambda)

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : z}{\Gamma : e\ x \Downarrow \Theta : z}$$ (app)

$$\frac{\Gamma : e \Downarrow \Theta : z}{(\Gamma, x \mapsto e) : x \Downarrow (\Theta, x \mapsto z) : \hat{z}}$$ (var)

$$\frac{(\Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) : e \Downarrow \Delta : z}{\Gamma : let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \Downarrow \Delta : z}$$ (let)

$\hat{z}$ means rename the bound variables in the value to be fresh

# Numbers

Arithmetic primitives are strict.

$$
\begin{array}{lcl}
n & \in & \mathit{Number} \\
\oplus & \in & \mathit{Primitive} \\
e \in \mathit{Exp} & ::= & n \\
 & \mid & e_1 \oplus e_2
\end{array}
$$

$$\Gamma : n \Downarrow \Gamma : n$$

$$\frac{\Gamma : e_1 \Downarrow \Delta : n_1 \qquad \Delta : e_2 \Downarrow \Theta : n_2}{\Gamma : e_1 \oplus e_2 \Downarrow \Theta : n_1 \oplus n_2}$$

# Constructors and constants

Case selection is strict.

$$\begin{array}{lcl} c & \in & \mathit{Constructor} \\ e \in \mathit{Exp} & ::= & c\ x_1 \ldots x_n \\ & | & \mathit{case}\ e\ \mathit{of}\ \{c_i\ y_1 \ldots y_{m_i} \rightarrow e_i\}_{i=1}^{n} \end{array}$$

$$\Gamma : c\ x_1 \ldots x_n \Downarrow \Gamma : c\ x_1 \ldots x_n$$

$$\frac{\Gamma : e \Downarrow \Delta : c_k\ x_1 \ldots x_{m_k} \qquad \Delta : e_k[x_i/y_i]_{i=1}^{m_k} \Downarrow \Theta : z}{\Gamma : \mathit{case}\ e\ \mathit{of}\ \{c_i\ y_1 \ldots y_{m_i} \rightarrow e_i\}_{i=1}^{n} \Downarrow \Theta : z}$$

# Other extensions and applications

Garbage collection

augment judgement with set of "active" names

add rule to remove non-"reachable" bindings from heap

Cost counting

augment judgement with reduction counter

Verification

use semantics to prove correctness of transformations

# Downloads

These slides and a Scala implementation of the semantics can be downloaded from:

http://code.google.com/p/kiama/wiki/Research