Boa Calculus

Barry Jay
Centre for Quantum Computing and Intelligent Systems
School of Software
University of Technology Sydney
Barry.Jay@uts.edu.au

December 16, 2013

Abstract

Boa calculus combines the best features of lambda calculus, pattern calculus and factorisation calculus into a single system.

Intensional Computation

Lambda calculus is **extensional**, being driven by a uniform account of the input-output behaviour of functions.

Pattern calculus is also **intensional**, being driven by a uniform account of internal structure, e.g.

polymorphic update functions

$$\texttt{update...} : \forall D. D \to D$$

where D represents an arbitrary data structure or database.

Four Approaches

	abstraction	query data	query funs
lambda calculus	yes		
pattern calculus	yes	yes	
factorisation calculus		yes	yes
boa calculus	yes	yes	yes

Program analysis queries the internal structure of **functions**.
Pattern calculus cannot do this; it queries data structures only.
Factorisation calculus queries anything, but no closures.
boa calculus queries lambda abstractions during evaluation.

A Core of Boa Calculus

$$
\begin{array}{rcl}
O & ::= & S \mid K \mid R \mid G \\
t & ::= & x \mid O \mid t\ t \mid \lambda x.t\ .
\end{array}
$$

- S and K are traditional.
- R is used to extract free variables from abstractions.
- G is a factorisation operator (like F of SF-calculus).

Represent I by SKK.
(In practice, add I, Y, constructors and E to support typed recursion, constructed data and equality of operators.)

Auxiliary concepts

The *values* are partially applied operators and abstractions, i.e.

$$v ::= S \mid St \mid Stu \mid K \mid Kt \mid R \mid Rt \mid G \mid Gt \mid Gtu \mid \lambda x.t$$

The *compounds* are all values that are applications.
The *eager terms* are those of the form

$$a, b ::= Stu \mid Rt \mid Gtu$$

(but **not** Kt or $\lambda x.s$).

The *at-terms* are those of the form $\quad a\ x \quad$ where a is eager.

Reduction

There are three sorts of reduction rules: beta rules, operator rules and at-rules (so boa calculus).
Beta reduction is limited to values and variables.

$$\begin{array}{rcl} (\lambda y.s)x & \rightharpoonup & \{x/y\}s \\ (\lambda y.s)v & \rightharpoonup & \{v/y\}s\,. \end{array}$$

S, R and G are call-by-value: K is call-by-name.

$$\begin{array}{rcl} Stuv & \longrightarrow & t\ v\ (u\ v) \\ Ktu & \longrightarrow & t \\ Rtv & \longrightarrow & \lambda x.t\ x\ v \quad (x \text{ fresh}) \\ GtuO & \longrightarrow & u \\ Gtu(qr) & \longrightarrow & t\ q\ r \quad (qr \text{ a compound}) \end{array}$$

Note that $Stux$ and Rtx are head normal.
The at-rules will be motivated by the analysis of abstractions.

Manipulating a head variable

Is x free or bound in

$$\lambda y_1. \ldots . \lambda y_n . x \; u_1 \; \ldots u_k \quad ?$$

First, push the head variable to the right using the rules

$$\begin{array}{rcl} x \; u & \longrightarrow & SI(Ku)x \\ a \; x \; u & \longrightarrow & Sa(Ku)x \end{array}$$

that produce at-terms, so that

$$x \; u_1 \; \ldots u_k \longrightarrow S(S \ldots (SI(Ku_1)) \ldots)(Ku_k)x \; .$$

Substitution of a value v for x **reverses** reduction, as in

$$a \; v \; u \longleftarrow a \; v \; (Kuv) \longleftarrow Sa(Ku)v \; .$$

Strange, but enough for confluence.

Applying functions to at-terms

Add the rules

$$\begin{array}{rcll} (\lambda y.s)(a\ x) & \rightharpoonup & S(K(\lambda y.s))a\ x & (a \text{ eager}). \\ b(a\ x) & \rightharpoonup & S(Kb)a\ x & (a, b \text{ eager}). \end{array}$$

This is enough, since all normal forms will be variables, at-terms, operators or compounds, once the rules for abstractions are added.

Analysing abstractions

$$
\begin{array}{rcl}
\lambda x.x & \longrightarrow & I \\
\lambda x.y & \longrightarrow & R(KI)y \\
\lambda x.a\ x & \longrightarrow & S(\lambda x.a)I \\
\lambda x.a\ y & \longrightarrow & R(\lambda x.a)y \\
\lambda x.O & \longrightarrow & KO \\
\lambda x.q\ r & \longrightarrow & S(\lambda x.q)(\lambda x.r) \quad \text{(if } q\ r \text{ is a compound.)}
\end{array}
$$

There is little room for modification of the constraints.
For example, substituting v for y in the fourth rule yields

$$\lambda x.a\ v \longleftarrow \lambda x.(\lambda x.a)x\ v \longleftarrow R(\lambda x.a)v$$

by beta reduction, so need beta for variables as well as values, but beta for at-terms would break confluence.

An Example

$$
\begin{array}{rcl}
\lambda x.x\ y & \longrightarrow & \lambda x.SI(Ky)\ x \\
 & \longrightarrow & S(\lambda x.SI(Ky))I \\
 & \longrightarrow & S(S(\lambda x.SI)(\lambda x.Ky))I \\
 & \longrightarrow & S(S(\lambda x.SI)(S(\lambda x.K)(\lambda x.y)))I \\
 & \longrightarrow & S(S(\lambda x.SI)(S(\lambda x.K)(R(KI)y)))I \\
 & \ldots &
\end{array}
$$

Elimination of a lambda is linear in the number of applications and so is exponential in the number of nested lambdas.

Equality of closed normal lambda abstractions is now decidable.

Future Work

- Formal proofs of confluence and progress (learning Coq now).
- Constraint types (for typing the equality operator).
- Typed self-interpreters (with Jens Palsberg).
- Program analysis (with Neil Jones).
- Abstract machine (with Jose Vergara).

Conclusions

boa calculus combines the best of extensional and intensional calculi by adding some operators to lambda calculus.

- supports (eager) beta-reduction (so, closures)
- supports queries of arbitrary closed normal forms (so, select and update of abstractions)
- aims at (typed) program analysis in the source language.
- Earlier versions required mysterious at-terms $t@x$ but these simplify to $a\ x$ with a eager.