An implementation of intensional computation

An abstract machine for the boa calculus

Jose Alberto Vergara Medina
(On joint work w. Barry Jay)

University of Technology Sydney

December 15, 2013

Implementing intensional computation

- Intensional computation is explicit in abstract machines (such as Turing Machines), since they can already have access to the internal structure of programs, so it is straightforward to implement.

boa-calculus

The term syntax of the boa calculus is given by

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

The atam

This presentation shows the development of an abstract machine (The ATAM), that implements the boa-calculus.

$$
\begin{array}{rcl}
O & ::= & c \mid S \mid K \mid I \mid Y \mid R \mid G \mid E \\
i & ::= & acc(n) \mid O \mid ap \mid mkclo \mid ret \mid abs \mid equal \mid fact \\
v & ::= & cl(t,e) \mid th(t,e) \mid op(O,e) \mid t@n \\
e & ::= & \bullet \mid v.e \mid \Uparrow e
\end{array}
$$

Intensional computation is supported by:

1. Adding partially applied operators and at-terms as values: $op(O,e)$, and $t@n$.
2. Adding three new instructions for intensional computation: *abs*, *equal* and *fact*.

A lambda calculus machine

$$\begin{array}{rcl} i & ::= & acc(n) \mid ap \mid mkclo \mid ret \\ v & ::= & cl(t,e) \mid th(t,e) \\ e & ::= & \bullet \mid v.e \end{array}$$

The *compilation* $[\![-]\!]$ of pure lambda terms is given by

$$\begin{array}{rcl} [\![n]\!] & = & acc(n) \\ [\![t\ \ u]\!] & = & [\![t]\!]; [\![u]\!]; ap \\ [\![\lambda t]\!] & = & mkclo[\![t]\!]\ . \end{array}$$

A lambda calculus machine

code	env	stack	code'	env'	stack'
$acc(n); c$	e	s	c	e	$e[n].s$
$ap; c$	e	$v.th(c_1, e_1).s$	c_1	e_1	$v.th(c, e).s$
$ap; c$	e	$v.cl(c_1, e_1).s$	c_1	$v.e_1$	$th(c, e).s$
$mkclo(c_1); c$	e	s	c	e	$cl(c_1, e).s$
ret	e	$v_1.v_2.s$	ap	e	$v_1.v_2.s$

Extensional and intensional operations

oper	*code*
S	$acc(2); acc(0); ap; acc(1); acc(0); ap; ap$
K	$acc(1)$
I	$acc(0)$
B	$acc(2); acc(1); acc(0); ap; ap$
C	$acc(2); acc(0); ap; acc(1); ap$
R	$mkclo[C]$
E	$acc(0); acc(1); equal$
G	$acc(0); fact$
$F2$	$mkclo(acc(0); acc(3); ap; acc(2); ap)$

Abstraction

code	env	stack		code'	env'	stack'
$abs; c$	e	$th(c_1, e_1).s$		c_1	e_1	$th(abs, id).th(c, e).s$
$abs; c$	e	$cl(c_1, e_1).s$		c_1	$\Uparrow e_1$	$th(AA, id).th(c, e).s$
$abs; c$	e	$O.s$		c	e	$KO.s$
$abs; c$	e	$op(O, v.e_1).s$		$A2$	e	$op(O, e_1).S.v.th(c, e).s$
$abs; c$	e	$v@0.s$		$A0$	id	$v.th(c, e).s$
$abs; c$	e	$v@n.s$	$n \geq 1$	$A1$	id	$v.I@(n-1).th(c, e).s$

Factorization

code	env	stack		code'	env'	stack'
$fact; c$	e	$th(c_1, e_1).s$		c_1	e_1	$th(fact, id).th(c, e).s$
$fact; c$	e	$cl(c_1, e_1).s$		c_1	$\Uparrow e_1$	$th(AF, id).th(c, e).s$
$fact; c$	e	$O.s$		c	e	$K.s$
$fact; c$	e	$op(O, v.e_1).s$		c	e	$cl(F2, v.op(O, e_1)).s$
$fact; c$	e	$(v@n).s$		c	e	$op(B, v, F)@n.s$

Example

Consider $GIS(\lambda y.\lambda x.x\ y)$. In deBruin indices this becomes $GIS(\lambda\ 0\ 1)$ which compiles to:

$$[\![GIS(\lambda\ 0\ 1)]\!] = G; I; ap; S; ap; mkclo(mkclo(acc(0); acc(1); ap; ret); ret); ap; ret$$

whose execution passes through the following states

code	env	stack
ap	Id	$Cl(acc(0); acc(1); ap; ret) \cdot G[I, S].s$
$abs; factor; return$	Id	$Cl(acc(0); acc(1); ap; ret).S.I.s$
$abs; abs; return$	Id	$C[I, v1]@0.Th(77).S.I.s$
$ap; return$	Id	$R[K[I]]@0.S[S[K[C].K[I]]].Th(abs0Val).Th(AA).Th(AG).S.I.s$

Example

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

Conclusions

Intensional computation is implicit in abstract machines since we can examine the environment of values to support factorization. In particular by adding three new instructions: *abs*,*fact* and *equal* for intensional computation. And two new notions of values: partially applied operators and at-terms.