

A Brief Survey of Abstract Machines for Functional Programming Languages

Prashant Kumar

University of Calgary

pkumar@ucalgary.ca

June 6, 2014

Abstract Machines for Functional Programming languages

Some abstract machines for functional languages are as follows:-

- Modern SECD
- Krivine Machine

Modern SECD Machine

- 1 An improvement over the SECD machine .
- 2 Doesn't have a dump . Stack doubles up as a dump.

Features of Modern SECD

- 1 a code pointer c (the instructions yet to be executed)
- 2 a stack s (holding intermediate result and pending function calls)
- 3 an environment e (giving values to the variables)
- 4 Modern SECD uses De-Bruijn's Indices to look into the environment.

Example

Lets consider $(\lambda x.xx)$ $(\lambda x.x)$ and $(\lambda x.\lambda y.(xy))$ $(\lambda x.x)$ $(\lambda y.y)$

De Bruijn's notation for the above lambda expression are as below

$$(\lambda x.xx) (\lambda x.x) \Rightarrow (\lambda.(\#1 \#1))(\lambda. \#1)$$
$$(\lambda x.\lambda y.(xy)) (\lambda x.x) (\lambda y.y) \Rightarrow (\lambda. \lambda.(\#2 \#1))(\lambda.\#1)(\lambda.\#1)$$

The De Bruijn Indices have been assumed to start from 1 here.

Data structures for Lambda and De Bruijn Lambda expressions

```
-- Basic lambda expressions ...
data Lambda = L_CONST Int
            | L_ADD Lambda Lambda
            | L_MUL Lambda Lambda
            | L_APP Lambda Lambda
            | L_ABST String Lambda
            | L_VAR String
            deriving (Eq, Show)

-- De Bruijn notation
data DBLambda = DB_CONST Int
              | DB_ADD DBLambda DBLambda
              | DB_MUL DBLambda DBLambda
              | DB_APP DBLambda DBLambda
              | DB_ABST DBLambda
              | DB_VAR String
              | DB_I Int -- de Bruijn index
              deriving (Eq, Show)

data SF a = FF | SS a
          deriving (Eq, Show)
```

Haskell code for converting Lambda expressions to De-Brujin Lambda expressions

```
lambda_2_db :: Lambda -> DBLambda
lambda_2_db l = db_translate [] l where
  db_translate env (L_CONST n) = DB_CONST n
  db_translate env (L_ADD l1 l2) = DB_ADD l1' l2' where
    l1' = db_translate env l1
    l2' = db_translate env l2
  db_translate env (L_MUL x1 x2) = DB_MUL l1' l2' where
    l1' = db_translate env x1
    l2' = db_translate env x2
  db_translate env (L_ABST v l) = DB_ABST l' where
    l' = db_translate (v:env) l
  db_translate env (L_APP l1 l2) = DB_APP l1' l2' where
    l1' = db_translate env l1
    l2' = db_translate env l2
  db_translate env (L_VAR v)
    = case (find v env) of
        SS n -> DB_I n
        FF -> DB_VAR v
    where
      find v [] = FF
      find v (v':rest) | v == v' = SS 1
                       | otherwise = case find v rest of
                                      FF -> FF
                                      SS n -> SS (n+1)
```

Instruction Set

Const - push the constant on the stack

Clo(c) - push closure of code c with current environment

Ret - terminate current function, jump back to caller

App - pop function closure and argument, perform application

Add - pop two arguments from the top of the stack and add

Access(n) - push n-th field of the environment

Compilation Scheme of Modern SECD Machine

Compilation Scheme

Lambda Terms	Compilation
$\llbracket k \rrbracket$	$Const(k)$
$\llbracket \lambda a \rrbracket$	$Clo(\llbracket a \rrbracket : Ret)$
$\llbracket MN \rrbracket$	$\llbracket N \rrbracket : \llbracket M \rrbracket : App$
$\llbracket a + b \rrbracket$	$\llbracket b \rrbracket : \llbracket a \rrbracket : Add$
$\llbracket \#n \rrbracket$	$Access(n)$

Example of Compilation

Compilation

Lets have a look at the compilation of source term:

$$(\lambda x .x + 1) 2 \Rightarrow (\lambda .\#1 + 1) 2$$

Compiled Code:

$$\text{Const}(2) : \text{Clo}[\text{Const}(1) : \text{Access}(1) : \text{Add} : \text{Ret}] : \text{App}$$

Data structures for Code , Environment and Stack in Haskell

```
-- stack S can have either an integer or a closure
data Stack = SVal Int
            | SClo [Command] [Env]
            deriving (Eq, Show)

-- Environment E can have either an integer or a closure
data Env = EVal Int
          | EClo [Command] [Env]
          deriving (Eq, Show)

-- command stack can only have the below commands
data Command = Const Int
              | Clo [Command]
              | App
              | Ret
              | Add
              | Access Int
              deriving (Eq, Show)
```

Haskell Code for Compilation to Modern SECD Commands

```
-- compiling from De Bruijn
mSEC_compile :: DBLambda -> [Command]

mSEC_compile (DB_CONST n) = [Const n]

mSEC_compile (DB_ABST l) = [Clo (c ++ [Ret])] where
    c = mSEC_compile l

mSEC_compile (DB_APP l1 l2) = c1 ++ c2 ++ [App] where
    c1 = mSEC_compile l1
    c2 = mSEC_compile l2

mSEC_compile (DB_ADD l1 l2) = c2 ++ c1 ++ [Add] where
    c1 = mSEC_compile l1
    c2 = mSEC_compile l2

mSEC_compile (DB_I i) = [Access i]
```

Machine Transitions in Modern SECD Machine

Before			After		
Code	Env	Stack	Code	Env	Stack
$Const\ k : c$	e	s	c	e	$k : s$
$Clo(c') : c$	e	s	c	e	$Clos(c', e) : s$
$App : c$	e	$Clos(c', e') : v : s$	c'	$v : e'$	$Clos(c, e) : s$
$Ret : c$	e	$v : Clos(c', e') : s$	c'	e'	$v : s$
$Add : c$	e	$n : m : s$	c	e	$(n + m) : s$
$Access(n); c$	e	s	c	e	$e(n) : s$

$Clos(c, e)$ denotes closure of code “ c ” with environment “ e ”

Start and End conditions in Modern SECD Machine

Initial State

$Code = c ; Environment = Nil ; Stack = Nil$

Final State

$Code = Nil ; Environment = Nil ; Stack = v$

Implementation of Transition Function in Haskell

```
-- Modern SECD step function
step :: ([Command],[Env],[Stack]) -> ([Code],[Env],[Stack])

step ((Const n):c,e,s) = (c,e,(SVal n):s)

step ((Clo c):c',e,s) = (c',e,(SClo c e):s)

step (App:c,e,(SClo c' e'):(SVal i):s) = (c',(EVal i):e',(SClo c e):s)

step (App:c,e,(SClo c' e'):(SClo c'' e''):s) = (c',(EClo c'' e''):e',(SClo c e):s)

step (Ret:_ ,e,w:(SClo c e'):s) = (c,e',w:s)

step (Add:c,e,(SVal x):(SVal y):s) = (c,e,(SVal(x+y)):s)

step ((Access n):c,e,s) = (c,e,w:s) where
    w = get n e
    get 1 ((EVal i):vr) = SVal i
    get 1 ((EClo c e):vr) = SClo c e
    get n (v:vr) = get (n-1) vr
```

Example of evaluation in Modern SECD Machine

Lets try evaluating $Const(2) : Clo[Const(1) : Access(1) : Add : Ret] : App$
which is compilation of $(\lambda x . x + 1) 2$

Let $c = Const(1) : Access(1) : Add : Ret$

Code	Env	Stack
$Const(2) : Clo(c) : App$	Nil	Nil
$Clo(c) : App$	Nil	$Const\ 2$
App	Nil	$Clos(c, Nil) : 2$
$Const(1) : Access(1) : Add : Ret$	2	$Clos(Nil, Nil)$
$Access(1) : Add : Ret$	2	$1 : Clos(Nil, Nil)$
$Add : Ret$	2	$2 : 1 : Clos(Nil, Nil)$
Ret	2	$3 : Clos(Nil, Nil)$
Nil	Nil	3

Instruction Sets in Krivine Machine

- 1 Constant
- 2 Grab
- 3 Access(n)
- 4 Push(c)

Compilation Scheme

Lambda Terms	Compilation
$\llbracket k \rrbracket$	<i>Constant</i> k
$\llbracket \lambda a \rrbracket$	<i>Grab</i> : $\llbracket a \rrbracket$
$\llbracket \#n \rrbracket$	<i>Access</i> (n)
$\llbracket MN \rrbracket$	<i>Push</i> ($\llbracket N \rrbracket$) : $\llbracket M \rrbracket$

Krivine Data Structures in Haskell

```
-- Krivine machine commands.  
-- Only commands can be there on the command stack.  
data Command = Const Int  
             | Grab  
             | Access Int  
             | Push [Command]  
             deriving (Eq , Show)  
  
-- Environment and Stack can only have Closures as shown below.  
data Closure = CLS [Command] [Closure]  
             deriving (Eq , Show)
```

Haskell Code for Compilation to Krivine Commands

```
-- Compiling to Krivine commands
krivine_compile :: DBLambda -> [Command]

krivine_compile(DB_CONST k) = [Const k]

krivine_compile (DB_I n) = [Access n ]

krivine_compile (DB_APP dbl1 dbl2) = [Push c2] ++ c1
                                     where c1 = krivine_compile dbl1
                                             c2 = krivine_compile dbl2

krivine_compile (DB_ABST dbl ) = [Grab] ++ (krivine_compile dbl)
```

Example of Compilation in Krivine Machine

Lets consider $(\lambda x.xx)(\lambda x.x)$

De Bruijn's notation for the above lambda expression is
 $(\lambda.(\#1\#1))(\lambda.\#1)$

$\llbracket (\lambda.(\#1\#1))(\lambda.\#1) \rrbracket$

$Push(\llbracket \lambda.\#1 \rrbracket) : \llbracket \lambda.(\#1\#1) \rrbracket$

$Push(Grab, \llbracket \#1 \rrbracket) : Grab : \llbracket (\#1\#1) \rrbracket$

$Push(Grab, Access(1)) : Grab : Push(\llbracket \#1 \rrbracket) : \llbracket \#1 \rrbracket$

$Push(Grab, Access(1)) : Grab : Push(Access(1)) : Access(1)$

Machine Transitions in Krivine Machine

Before			After		
Code	Env	Stack	Code	Env	Stack
$Access(1) : c$	$Clos(c_0, e_0) : e$	s	c_0	e_0	s
$Access(n+1) : c$	$Clos(c_0, e_0) : e$	s	$Access(n)$	e	s
$Push(c') : c$	e	s	c	e	$Clos(c', e)$
$Grab : c$	e	$Clos(c', e') : s$	c	$Clos(c', e') : e$	s

Start and End conditions in Krivine Machine

Initial State

$Code = c; Environment = Nil; Stack = Nil$

Final State

$Code = Grab : c; Environment = Nil; Stack = Nil$

The code is decompiled to get the result.

Haskell Code for Implementation of the Transition function of krivine Machine

```
type Kmachine = ([Command],[Closure],[Closure])

eval :: Kmachine -> ([Command],[Closure])
eval (Grab:c,e,[]) = (Grab:c,e) -- ending condition
eval st = eval (step st )

step :: Kmachine -> Kmachine -- transition function
step ((Access 1):cs,(CLS c e):es ,s) = (c, e, s)
step ((Access n):cs,(CLS c e):es, s) = (Access (n-1):cs, es, s)
step ((Push cs'):cs, es, s) = (cs,es,(CLS cs' es):s)
step (Grab:cs, es, e:s) = (cs, e:es, s)
```

Example of evaluation in Krivine Machine

Lets consider the λ -expression $(\lambda x.xx)(\lambda x.x)$.

This translates to the following.

$Push(Grab, Access(1)) : Grab : Push(Access(1)) : Access(1)$

Let P , G and A stand for $Push$, $Grab$ and $Access$ respectively.

Code	Env	Stack
$P(G : A(1)) : G : P(A(1)) : A(1)$	<i>Nil</i>	<i>Nil</i>
$G : P(A(1)) : A(1)$	<i>Nil</i>	$Clos((G : A(1)), Nil)$
$P(A(1)) : A(1)$	$Clos((G : A(1)), Nil)$	<i>Nil</i>
$A(1)$	$Clos((G : A(1)), Nil)$	$Clos(A(1), Clos((G : A(1)), Nil))$
$G : A(1)$	<i>Nil</i>	$Clos(A(1), Clos((G : A(1)), Nil))$
$A(1)$	$Clos(A(1), Clos((G : A(1)), Nil))$	<i>Nil</i>
$A(1)$	$Clos((G : A(1)), Nil)$	<i>Nil</i>
$G : A(1)$	<i>Nil</i>	<i>Nil</i>

$G; A(1)$ decompiles to $\lambda .1$ which is $\lambda x.x$