

COMPUT325: SECD Virtual Machine

Dr B. Price and Dr. R. Greiner

4th November 2004

Real Functional Languages

- ▶ λ -calculus defines the semantics of functional languages
- ▶ λ -calculus (and therefore any abstraction of λ -calculus like pure Lisp) can be implemented in λ -calculus
- ▶ But how can we practically implement λ -calculus or another functional language on real hardware
- ▶ The basic unit of representation in a digital computer is not the λ -function

The SECD Machine

- ▶ Java Virtual Machine implements simple underlying operations for imperative and object-oriented languages
- ▶ Simple Machine implemented on dozens of platforms
- ▶ SECD machine is a virtual machine for functional languages
 - ▶ used in many implementations (LispMe for Palm Pilot)
- ▶ SECD implements
 - ▶ primitives for
 - ▶ values like integers - represented by bits as usual
 - ▶ composite structures like cons cells - represented by 2-element pointer vectors and
 - ▶ four special internal registers to represent computation state
 - ▶ operations to carry out computations
 - ▶ heap of memory cells



Stacks

- ▶ The SECD is a stack-based computer (like postscript or fourth)
- ▶ Stacks are represented as a list
 - ▶ $L = (s1\ s2\ s3\ s4\ \dots\ sn)$
- ▶ A dot in a list introduces its tail
 - ▶ Let $R = (s2\ s3\ s4\ \dots\ sn)$
 - ▶ Then $L = (s1\ .\ R)$
- ▶ We can easily refer to the first m elements of a stack as
 - ▶ $(s1\ s2\ \dots\ sm\ .\ <rest>)$
notice how the dot is used!



The SECD Stacks

- ▶ 4 Special registers point to 4 stacks
 - ▶ S=Scratch (for operands of operations and evaluated results)
 - ▶ E=Environment (stack of variable bindings in force)
 - ▶ C=Code (stack of primitive operations to execute in the active function)
 - ▶ D=Dump (stack of suspended computations)
 - each suspended computation has
 - ▶ a stack,
 - ▶ environment and
 - ▶ code body (S,E,C)
- ▶ Items stored in stacks may be atoms, or lists

Simplified Example

- ▶ A sample scratch stack with operands for PLUS:
S=(1 2 . rest)
- ▶ Result of PLUS is left in place of the operands
S=(3 . rest)

The SECD Machine Operations

- ▶ The state of the SECD machine is determined by the *four* stack registers
- ▶ SECD Operations transform the stacks from one state to another
- ▶ Legal transformations are defined by *rewrite rules*
- ▶ When the left side of the rule matches the state of the machine, the machine switches to the state given by the right side of the rule

$$s\ e\ c\ d \rightarrow s'\ e'\ c'\ d'$$

Simple SECD Program I

- ▶ Program = <list of primitive functions> + <immediate operands>
- ▶ A simple program to load the constants 3 and 5 onto the *scratch* stack
 - (LDC 3 LDC 5)
 - ▶ Here, LDC is the primitive function "Load Constant"
 - ▶ And 3 is an immediate operand
- ▶ The machine starts with the program loaded on the *code* stack

$$(s\ e\ c\ d) = (nil\ nil\ (LDC\ 3\ LDC\ 5).\ nil\ nil)$$

- ▶ Programs are processed one operation at a time using rewrite rules

Simple SECD Program II

- ▶ The rewrite rule for LDC is

$$s \ e \ (\text{LDC } x \ . \ c) \ d \rightarrow x.s \ e \ c \ d$$

- ▶ The constant x is pushed onto the front of the `scratch stack s`
- ▶ The `LDC x` operation is popped off of the code stack, leaving its tail c
- ▶ Execution of our simple program yields:

$$\begin{array}{l} s \quad \quad e \ (\text{LDC } 3 \ \text{LDC } 5) . c \ d \\ 3.s \quad \quad e \ (\text{LDC } 5) . c \quad \quad d \\ (5 \ 3).s \ e \ c \quad \quad \quad \quad d \end{array}$$



Additional Operations

- ▶ Typical arithmetic operations: ADD, SUB, MUL, DIV, REM, etc.

addition:

$$\begin{array}{l} (m \ n \ . \ s) \ e \ (\text{ADD} \ . \ c) \ d \\ \rightarrow (p \ . \ s) \ e \ c \quad \quad \quad d \quad ; ; \text{ where } p=m+n \end{array}$$

- ▶ Relational functions such as $=$ $>$ $<$ are also defined

compare:

$$\begin{array}{l} (m \ n \ . \ s) \ e \ (> \ . \ c) \ d \\ \rightarrow (b \ . \ s) \ e \ c \quad \quad \quad d \quad ; ; \text{ where } b=T \text{ if } m>n \text{ else } b=F \end{array}$$



More Complex Example

```
→s          e (LDC 3 LDC 5 ADD LDC 10 >) d
→3.s        e (LDC 5 ADD LDC 10 >) d
→(5 3).s    e (ADD LDC 10 >) d
→8.s        e (LDC 10 >) d
→10 8 .s    e (>) d
→T .s       e nil d
```

Branching : SEL and JOIN

- ▶ IF statement functionality is implemented with the *select* and *join* operations.
- ▶ Select (SEL)
 - ▶ chooses between two subprograms and
 - ▶ suspends remainder of main program by putting it on the dump stack
- ▶ Let T be 'true' and F be 'false'
 - ▶ IF T is on the stack, do subprogram $\langle C1 \rangle$ store rest of code **cr**

```
(T . s) e (SEL  $\langle C1 \rangle$   $\langle C2 \rangle$  . cr) d ;; s e c d
→s e  $\langle C1 \rangle$  (cr . d)
```

- ▶ IF F is on the stack, do subprogram $\langle C2 \rangle$ store rest of code **cr**

```
(F . s) e (SEL  $\langle C1 \rangle$   $\langle C2 \rangle$  . cr) d
→s e  $\langle C2 \rangle$  (cr . d)
```

"Un-branching" : JOIN

- ▶ Join restores the suspended main program from the dump stack
- ▶ SEL and JOIN work together to implement "IF" behaviour
- ▶ An example of an abstract IF and the equivalent SECD code:

```

IF 5 > 3
    THEN m-n
    ELSE 0
LDC 8
≡ (LDC 3 LDC 5 > SEL
;; SEL applied to result of 3 > 5
    (LDC m LDC n SUB JOIN)
    (LDC 0 JOIN)
    LDC 8)

```

- ▶ Unlike assembler, programs may have nested structures

Complex Branching Example

```

s      e (LDC 3 LDC 5 > SEL (LDC m LDC n SUB JOIN)
              (LDC 0 JOIN) LDC 8).c d
3.s    e (LDC 5 > SEL (LDC m LDC n SUB JOIN)
              (LDC 0 JOIN) LDC 8).c d
5 3.s  e (> SEL (LDC m LDC n SUB JOIN)
              (LDC 0 JOIN) LDC 8).c d
T.s    e (SEL (LDC m LDC n SUB JOIN)
              (LDC 0 JOIN) LDC 8).c d
s      e (LDC m LDC n SUB JOIN) ((LDC 8).c d)
m.s    e (LDC n SUB JOIN) ((LDC 8).c d)
(n m).s e (SUB JOIN) ((LDC 8).c d)
p.s    e (JOIN) ((LDC 8).c d) ;; where p = n-
m
p.s    e (LDC 8).c d ;; where p = n-m
(8 p).s e c d ;; where p = n-m

```

List Operations

NIL adds nil to scratch stack

```
      s e (NIL . c) d →  
(NIL . s) e c          d
```

CONS replaces top two elements of s with their consed pair

```
(x y . s) e (CONS . c) d →  
(x.y).s e c          d
```

CAR replaces top element with its CAR

```
(x.y) .s e (CAR . c) d →  
x .s e c          d
```

CDR replaces top element with its CDR

```
(x.y) .s e (CDR . c) d →  
y .s e c d
```



List Operators Example

```
      s          e (LDC 1 LDC 2 CONS CDR).c d  
→1.s          e (LDC 2 CONS CDR).c d  
→2 1.s        e (CONS CDR).c d  
→(2 1).s      e (CDR).c d  
→1.s          e c d
```



SECD User Defined Functions

- ▶ SECD, being stack based, put args on stack, then applies function
- ▶ The procedure for definition of user-functions is also backwards in spirit:
 - ▶ λ -calculus application: (`<function-def>``<arguments>`)
 - ▶ SECD application: `<arguments>``<function-def>` `AP`

SECD Function Definition Idiom

- ▶ Roughly
 - ▶ Construct list of function `arguments` on the scratch stack
 - ▶ Cons together loaded constants "LDC" or results of prior computations
 - ▶ Create closure = (`function` , `environment`) and put on scratch stack "LDF"
 - ▶ Eval closure using apply "AP"
 - ▶ saves current scratch, code and environment stacks
 - ▶ create fresh scratch stack for new function
 - ▶ make new environment from closure `environment` plus stack `arguments`
 - ▶ create code stack from closure `function`
 - ▶ when needed, copy arguments from env and put on scratch
 - ▶ do computation and leave results on scratch stack
 - ▶ Restore stacks

Creating Arguments on Scratch

- ▶ This is just ordinary list construction
- ▶ To pass the arguments 1, 2 to a user function, create list (1 2)

```
(NIL          ;; nil . s
LDC 2        ;; 2 nil . s
CONS         ;; (2) . s
LDC 1        ;; 1 (2) . s
CONS)        ;; (1 2) . s
```

- ▶ The argument list as a whole is typically called v
- ▶ We could represent scratch with arg list on top as: $v.s$

Load Function: LDF

- ▶ Create cons cell $(f . e)$ to hold closure on scratch stack
 - ▶ Copy function f from code to car of closure
 - ▶ Copy current environment e to cdr of closure

- ▶ The rewrite rule is:

$$v.s \quad e \quad (\text{LDF } f . c) \quad d \quad \rightarrow \quad ;; \quad s \quad e \quad c \quad d$$
$$((f.e) \quad v.s) \quad e \quad c \quad d$$

- ▶ Could create as many closures with environment e as desired

Apply Function: AP

- ▶ Saves current machine state onto dump stack
- ▶ Installs code from closure f into code stack
- ▶ creates new environment consisting of arguments from control stack v + environment saved in closure e'

$$\begin{array}{l} ((f.e') \ v.s) \quad e \quad (AP \ . \ c) \quad d \rightarrow \\ \text{NIL} \quad \quad \quad v.e' \quad f \quad \quad \quad s \ e \ c \ . \ d \end{array}$$

- ▶ Notice that the environment takes the form of a *list of lists* of values
- ▶ The first list, v , being the arguments and the second list, e' , being the lexical environment f was defined in
- ▶ The code for f has been put on the code stack
- ▶ The original scratch, env and code stacks are saved on dump



Retrieving Arguments and Variables

- ▶ The "LD" function retrieves values from environment
- ▶ Values are not retrieved by name, but by index in the list
- ▶ Like compiling an identifier to a "relative" memory address.
- ▶ Suppose the value 'x' is stored in slot j of nested environment i

$$\begin{array}{l} s \quad \quad \quad e \quad (LD \ (i.j).c) \quad d \rightarrow \\ (x.s) \quad e \quad c \quad \quad \quad d \quad ; \text{ where } x = \text{locate}((i.j), e) \end{array}$$

- ▶ Locate returns the the j^{th} value from the i^{th} list of environment e
- ▶ Retrieve j^{th} immediate argument with LD (1,j)
- ▶ Retrieve j^{th} variable in immediate environment with LD (2,j)
- ▶ Arguments reside in a local environment for the called function



Returning from a Function Call

- ▶ The RTN function restores the machine state on call completion
 - ▶ Copies *saved stacks* from dump back to original registers
 - ▶ Pushes returned value *x* from function onto top of restored stack

```
x.s' e' RTN.c' s e c . d →  
x.s e c d
```

- ▶ Returning function's stacks are discarded, except:
 - ▶ Value returned by function is copied to head of restored scratch
- ▶ Other stacks restored to pre-call state

Compiling a Function Application

- ▶ The square function applied to 3

```
(square 3)  
(NIL LDC 3 CONS ;; build arguments  
LDF ( LD (1.1) ;; code for square in a sublist  
LD (1.1) ;; code loads 2 copies of arg from er  
MUL ;; then multiplies  
RTN )  
AP) ;; apply square function
```

Evaluating a Function Call

- ▶ Let F= (LD (1.1) LD (1.1) MUL RTN) so that the application of square is

```
s          e (NIL LDC 3 CONS LDF F AP).c d ;; s e c d
nil.s     e (LDC 3 CONS LDF F AP).c      d
3 nil.s   e (CONS LDF F AP).c           d
(3).s     e (LDF F AP).c                d
(F.e) (3).s e (AP).c                    d
nil      (3).e F                        ((s e c) . d)
```

Evaluating Function Body and Returning

```
nil (3).e F ((s e c) . d)
```

```
RECALL:F= (LD (1.1) LD (1.1) MUL RTN)
```

```
nil (3).e (LD (1.1) LD (1.1) MUL RTN) (s e c . d)
```

```
3 (3).e ( LD (1.1) MUL RTN) (s e c . d)
```

```
3 3 (3).e (MUL RTN) (s e c . d)
```

```
9 (3).e (RTN) (s e c . d)
```

```
9.s e c d
```

Named Functions

- ▶ In the previous example we apply an immediate function
- ▶ Generally we want to apply named functions

Let `square(x) = x*x` IN `square(3)`

- ▶ This is equivalent to

`(λf | f(3)) (λx | x*x)`

Named Functions

- ▶ Repeated:

`(λf | f(3)) (λx | x*x)`

- ▶ Thus, we must apply the function body as an argument to a `λ` in order to name it

```
NIL
LDF (LD (1.1) LD (1.1) MUL RTN) ; square: <(λy|y*y),e>
CONS                               ; scratch: (<square.e>).s
LDF (
  NIL          LDC 3 CONS          ; arg on stack (3)
  LD (1 . 1)   ; retrieve closure <square.e>
  AP           ; app closure to arg
  RTN)
;; scratch: (λf | f 3) (<square.e>) .s
AP ;; apply f to (<square.e>)
```

Trace of Named Functions I

Let $X = (LD (1.1) LD (1.1) MUL RTN)$

Let $F = (NIL LDC 3 CONS LD (1 . 1) AP RTN)$

```
s           e           (NIL LDF X CONS LDF F AP).c    d
NIL.s      e           (LDF X CONS LDF F AP).c    d
(X.e) NIL.s e           (CONS LDF F AP).c    d
((X.e)).s  e           (LDF F AP).c    d ;; clo-
sure as arg
```

;; Now have closure and arguments on top of scratch stack

```
(F.e) ((X.e)).s e           (AP).c    d
nil      ((X.e) e) F      (s e c.d)
```

;; Note, closure for X is 1st value in first frame



Trace of Named Functions II

Recall $F = (NIL LDC 3 CONS LD (1 . 1) AP RTN)$

;; We omit dump parameter here:

```
nil      ((X.e) e) F
nil      ((X.e) e) (NIL LDC 3 CONS LD (1 . 1) AP RTN)

nil.nil  ((X.e) e) (LDC 3 CONS LD (1 . 1) AP RTN)

3 nil.nil ((X.e) e) (CONS LD (1 . 1) AP RTN)

(3).nil  ((X.e) e) (LD (1 . 1) AP RTN)
```

*;; We load closure from X from environ-
ment (ie the SQUARE function)*

```
(x.e) (3).nil ((X.e) e) (AP RTN)
```

;; Evaluation of square now proceeds as in the anonu-

Recursive Functions

- ▶ As in meta-interpretation of λ -calculus, we build a self-referencing closure

LETREC $f(x) = \langle \text{BODY} \rangle$ IN $f(y)$
 $\equiv (\lambda f | f(y)) \langle \text{BODY} \rangle$

- ▶ When we pass function body in, we use the closure mechanism

$C \equiv \langle \langle \text{BODY} \rangle, f \leftarrow C \rangle$ $E \equiv \langle f(v), f \leftarrow C \rangle$

Recursive Functions

- ▶ Build a self-referencing environment in two steps

- ▶ DUM creates an unused slot in the environment (NIL is used to fill the slot, but this is irrelevant)

$s \ e \ (DUM \ LDF \ F \ . \ c) \ d$
 $\rightarrow s \ NIL.e \ (LDF \ F \ . \ c) \ d$
 $\rightarrow (F.NIL.e) \ NIL.e \ c \ d$

- ▶ RAP (recursive apply) calls `rplaca` to assign slot to be v

$((f.NIL.e') \ v.s)$	$(NIL.e)$	$(RAP.c)$	$d \rightarrow$
NIL	$rplaca((NIL.e'), v)$	f	$(s \ e \ c.d)$
NIL	$(v.e')$	f	$(s \ e \ c.d)$

Recursive Length

```
(letrec (f (λx m | (if (null x) m (f (cdr x) (+ m 1) )) ) )
  (f '(1 2 3) 0) )
(DUM ;; (nil . e)
NIL LDF( ;; (λx m | ...
  LD (1.1) NULL SEL ;; if null x
  (LD (1.2) JOIN) ;; then return m
  ;; else
  (NIL LDC 1 ld (1.2) ADD CONS ;; form (q) where q=m+1
  LD (1.1) CDR CONS ;; form (z q) where z=(cdr x)
  LD (2.1) AP JOIN) ;; Apply f to (z q)
  RTN)
CONS ;; Arg list contains closure: ( (F.e) ) . s
LDF ;; (λf | ..
  (NIL LDC 0 CONS LDC (1 2 3) CONS LD (1.1) ;; (F (1 2 3) 0)
  AP RTN)
;; f v .s ≡ (λf.(nil . e)) ( (λx m.(nil . e)) ) .s
RAP) ;; ≡(rplca (nil . e) v), where v= ( (λx m.(nil . e)) )
```

Recursive Length Notes

- ▶ The key to the previous is example is the last two lines:

```
;; f v .s ≡ (λf.(nil . e)) ( (λx m.(nil . e)) )
RAP) ;; ≡(rplca (nil . e) v), where v= ( (λx m.(nil . e)) )
```
- ▶ Notice that when `nil` is replaced by `v= ((λx m.(nil . e)))`, the closure `(λx m.(nil . e))` has its first environment frame pointing back to itself
- ▶ When this closure is executed, the arguments the closure are called on will become the new first frame
- ▶ The self referencing point will become the first argument of the second frame (i.e., LD (2.1))

Recursive Functions: Fact

- ▶ Suppose we were interested in this code:

```
let x=3 and one = 1 in
  letrec f(n,m)=
    if (eq n 0) then one
    else f(n - one, n × m)
  in f(x,one)
```

- ▶ This translation is not quite right: `f` cannot refer to itself:

```
(λx,one |
  (λf|f(x,one))
  (λn m |
    if (eq n 0) then one else f( n - one, n × m))
) (3 1)
```

Recursive Fact in SECD Code

```
(nil ldc 1 cons ldc 3 cons      ;; (λx,one |⟨BODY⟩) (3 1)
ldf                             ;; adds fact closure to scratch
(dum                             ;; adds a null environment
nil                             ;; inner λ arg list (fact)
ldf                             ;; fact closure <f.e> -> scratch
  (ldc 0 ld (1,1) eq sel        ;; if 0=n
  (ldc 1 join)                  ;; then return 1
  (nil                          ;; else: recurse
  ld(1,2) ld(1,1) MPY CONS      ;; n × m
  LD (3,2) LD(1,1) SUB CONS     ;; n - 1
  LD(2,1) AP JOIN)             ;; load fact and apply
RTN)
CONS ;; create argument of closure (<f.e>)
LDF (NIL LD(2.2) CONS
  LD (2.1) CONS ;; <f,e> (x,one) . s
  LD (1.1)
  AP RTN) ;; <
RAP RTN) AP)
```

STOP

- ▶ Signals that computation should be halted
- ▶ Ending programs with STOP forces programmer to be explicit
- ▶ Allows virtual machine to signal an error if it runs out of instructions for some other reason

Practicalities

- ▶ We quickly run out of memory without garbage collection
- ▶ Variety of collection strategies with different properties:
 - ▶ Reference Count
 - ▶ Mark and Sweep
 - ▶ Generation Scavenging (Baker's algorithm)