Programs as Data 6
Imperative languages, environment and store, micro-C

Peter Sestoft
Monday 2012-10-01*
Today

• Course overview
• A naïve imperative language
• C concepts
  – Pointers and pointer arithmetics, arrays
  – Lvalue and rvalue
  – Parameter passing by value and by reference
  – Expression statements
• Micro-C, a subset of C
  – abstract syntax
  – lexing and parsing
  – interpretation
The overall course plan

- F# and functional programming
- Interpreting an expression language
- Lexing and parsing tools
- Interpreting a functional language, micro-ML
  - Higher-order functions
- Type checking and type inference
- Interpreting an imperative language, micro-C
- Compiling micro-C to stack machine code
- Real-world abstract machines: JVM and .NET
  - Garbage collection techniques
- Continuations, exceptions and backtracking
- (Programs that generate programs, Scheme)
- Or maybe Scala, a functional/OO language on JVM
A naive-store imperative language

- **Naive** store model:
  - a variable name maps to an integer value
  - so store is just a runtime environment

```plaintext
sum = 0;
for i = 0 to 100 do
  sum = sum + i;
```

<table>
<thead>
<tr>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5050</td>
</tr>
</tbody>
</table>

```plaintext
i = 1;
sum = 0;
while sum < 10000 do begin
  sum = sum + i;
  i = 1 + i;
end;
```

<table>
<thead>
<tr>
<th>i</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>10011</td>
</tr>
</tbody>
</table>
Naïve-store statement execution, 1

- Executing a statement gives a new store
- Assignment $x = e$ updates the store
- Expressions do not affect the store

```
let rec exec stmt (store : naivestore) : naivestore =
  match stmt with
  | Asgn(x, e) ->
    setSto store (x, eval e store)
  | If(e1, stmt1, stmt2) ->
    if eval e1 store <> 0 then exec stmt1 store
    else exec stmt2 store
  | ...
```

Update store at $x$ with value of $e$
Naïve-store statement execution, 2

- A block \( \{s_1; \ldots; s_n\} \) executes \( s_1 \) then \( s_2 \) ...

- Example:

\[
\text{exec (Block } [s_1; s_2]) \text{ store} \\
= \text{loop } [s_1; s_2] \text{ store} \\
= \text{exec } s_2 \text{ (exec } s_1 \text{ store)}
\]

```ocaml
let rec exec stmt (store : naivestore) : naivestore =
  match stmt with
  | Block stmts ->
    let rec loop ss sto =
      match ss with
      | [] -> sto
      | s1::sr -> loop sr (exec s1 sto)
    loop stmts store
  | ...```
Naïve-store statement execution, 3

- **for** and **while** update the store sequentially

```ocaml
let rec exec stmt (store : naivestore) : naivestore =
  match stmt with
  | ... |
  | For(x, estart, estop, stmt) -> ...
  | While(e, stmt) ->
    let rec loop sto =
      if eval e sto = 0 then sto
      else loop (exec stmt sto)
    in loop store
```

```ocaml```
Environment and store, micro-C

- The naive model cannot describe *pointers* and *variable aliasing*

- A more realistic store model:
  - *Environment* maps a variable name to an address
  - *Store* maps address to value

\[
\begin{array}{c}
\text{i:} & 42 \\
\text{sum:} & 44 \\
\end{array}
\]

\[
\begin{array}{ccccc}
41 & 42 & 43 & 44 & 45 \\
... & 100 & ... & 5050 & \\
\end{array}
\]
The essence of C: Pointers

• Main innovations of C (1972) over Algol 60:
  – Structs, as in COBOL and Pascal
  – Pointers, pointer arithmetics, pointer types, array indexing as pointer indexing
  – Syntax: {} for blocks, as in C++, Java, C#

• Very different from Java and C#, which have no pointer arithmetics, but garbage collection
## Desirable language features

<table>
<thead>
<tr>
<th>Feature</th>
<th>C</th>
<th>C++</th>
<th>F#/ML</th>
<th>Smtalk</th>
<th>Haskell</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Garbage collection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exceptions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounds checks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static types</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic types (para. polym.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern matching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reflection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Refl. on type parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anonymous functions (λ)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Streams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lazy eval.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C pointer basics

• A pointer \( p \) refers to a storage location
• The dereference expression \( *p \) means:
  - *the content of the location* (rvalue) as in \( *p + 4 \)
  - *the storage location itself* (lvalue), as in \( *p = x+4 \)
• The pointer that points to \( x \) is \&x
• Pointer arithmetics:
  *\((p+1)\) is the location just after *\( p \)
• If \( p \) equals \&a[0]
  then *\((p+i)\) equals \( p[i] \) equals \( a[i] \), so an array is a pointer
• Strange fact: \( a[2] \) can be written 2[a] too
Using pointers for return values

- Example ex5.c, computing square(x):

```c
void main(int n) {
    ... 
    int r;
    square(n, &r);
    print r;
}

void square(int i, int *rp) {
    *rp = i * i;
}
```

for input

for return value: a pointer to where to put the result
Recursion and return values

• Computing factorial with MicroC/ex9.c

```c
void main(int i) {
    int r;
    fac(i, &r);
    print r;
}

void fac(int n, int *res) {
    if (n == 0)
        *res = 1;
    else {
        int tmp;
        fac(n-1, &tmp);
        *res = tmp * n;
    }
}
```

• `n` is input parameter
• `res` is output parameter: a pointer to where to put the result
• `tmp` holds the result of the recursive call
• `&tmp` gets a pointer to `tmp`
Storage model for micro-C

• The store is an indexable stack
  – Bottom: global variables at fixed addresses
  – Plus, a stack of activation records

    | globals | main | fac(3) | fac(2) | fac(1) | fac(0) |
    |---------|------|--------|--------|--------|--------|

• An activation record is an executing function
  – return address and other administrative data
  – parameters and local variables
  – temporary results

    | admin. data | params+locals | temps |
Lvalue and rvalue of an expression

- Rvalue is “normal” value, right-hand side of assignment: 17, true
- Lvalue is “location”, left-hand side of assignment: x, a[2]
- In assignment e1=e2, expression e1 must have lvalue

<table>
<thead>
<tr>
<th></th>
<th>Has lvalue</th>
<th>Has rvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>a[2]</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>*p</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>x+2</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;x</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

- Where else must an expression have lvalue in C#? In C?
Call-by-value and call-by-reference, C#

```csharp
static void swapR(ref int x, ref int y) {
    int tmp = x; x = y; y = tmp;
}

static void swapV(int x, int y) {
    int tmp = x; x = y; y = tmp;
}

int a = 11;
int b = 22;
swapV(a, b);
swapR(ref a, ref b);
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>42</td>
</tr>
</tbody>
</table>

Addresses:

- `int x` is stored at address 43
- `int y` is stored at address 44
- `int tmp` is stored at address 45

By value:

```csharp
static void swapV(int x, int y) {
    int tmp = x; x = y; y = tmp;
}
```

By reference:

```csharp
static void swapR(ref int x, ref int y) {
    int tmp = x; x = y; y = tmp;
}
```
# C variable declarations

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>int n</td>
<td>n is an integer</td>
</tr>
<tr>
<td>int *p</td>
<td>p is a pointer to integer</td>
</tr>
<tr>
<td>int ia[3]</td>
<td>ia is array of 3 integers</td>
</tr>
<tr>
<td>int *ipa[4]</td>
<td>ipa is array of 4 pointers to integers</td>
</tr>
<tr>
<td>int (*iap)[3]</td>
<td>iap is pointer to array of 3 integers</td>
</tr>
<tr>
<td>int *(*ipap)[4]</td>
<td>ipap is pointer to array of 4 pointers to ints</td>
</tr>
</tbody>
</table>

Unix program **cdecl** or [www.cdecl.org](http://www.cdecl.org) may help:

```
cdecl> explain int *(*ipap)[4]
declare ipap as pointer to array 4 of pointer to int
ncdecl> declare n as array 7 of pointer to pointer to int
ncdecl> int **n[7]
```
Micro-C array layout

- An array `int arr[4]` consists of
  - its 4 int elements
  - a pointer to `arr[0]`

- This is the uniform array representation of B
- Real C treats array parameters and local arrays differently; complicates compiler
- Strachey’s CPL -> Richards’s BCPL -> B -> C
Micro-C syntactic concepts

• Types
  int
  int *x
  int x[4]
  TypI
  TypP(TypI)
  TypA(TypI, Some 4)

• Expressions
  (*p + 1) * 12

• Statements
  if (x!=0) y = 1/x;

• Declarations
  – of global or local variables
  int x;
  – of global functions
  void swap(int *x, int *y) { ... }
| Type typ =                              | (* Type int                    *) |
| | TypI                                  | (* Type char                   *) |
| | TypC                                  | (* Array type                  *) |
| | TypP of typ                            | (* Pointer type                *) |
| and expr =                              | (* x or *p or a[e]             *) |
| | Access of access                      | (* x    or  *p    or  a[e]     *) |
| | Assign of access * expr               | (* x=e or *p=e or a[e]=e        *) |
| | Addr of access                        | (* &x or &*p or &a[e]            *) |
| | CstI of int                           | (* Constant                    *) |
| | Prim1 of string * expr                | (* Unary primitive operator    *) |
| | Prim2 of string * expr * expr          | (* Binary primitive operator    *) |
| | Andalso of expr * expr                | (* Sequential and              *) |
| | Orelse of expr * expr                 | (* Sequential or               *) |
| | Call of string * expr list            | (* Function call f(...)        *) |
| and access =                            | (* Variable access        x    *) |
| | AccVar of string                      | (* Pointer dereferencing  *p   *) |
| | AccDeref of expr                      | (* Array indexing         a[e]  *) |
| and stmt =                              | (* Conditional                *) |
| | If of expr * stmt * stmt              | (* While loop                  *) |
| | While of expr * stmt                  | (* Expression statement   e;   *) |
| | Expr of expr                          | (* Return from method         *) |
| | Return of expr option                 | (* Block: grouping and scope  *) |
| and stmtordec =                         | (* Local variable declaration  *) |
| | Dec of typ * string                   | (* A statement                 *) |
| and topdec =                            | (* Fundec of typ option * string * (typ * string) list * stmt |
| | Stmt of stmt                          | (* Vardec of typ * string      *) |
| and program =                           | (* Prgm of topdec list          *) |
Lexer specification for micro-C

• New: endline comments  // blah blah
  and delimited comments  if (x /* y? */)

rule Token = parse
| ...                       { () }
| "//"                     { EndLineComment lexbuf; Token lexbuf }
| "/*"                     { Comment lexbuf; Token lexbuf }

and EndLineComment = parse
| ['\n' 'r']                { () }
| (eof | '\026')            { () }
| _                         { EndLineComment lexbuf }

and Comment = parse
| "/*"                     { Comment lexbuf; Comment lexbuf }
| "*/"                     { () }
| ['\n' 'r']                { Comment lexbuf }
| (eof | '\026')            { lexerError lexbuf "Unterminated" }
| _                         { Comment lexbuf }
Parsing C variable declarations

- Hard, declarations are *mixfix*: `int *x[4]`
- Parser trick: Parse a variable declaration as a type followed by a variable description:
  \[
  \text{int} \quad *x[4]
  \]

- Parse var description to get pair \((f, x)\) of type function \(f\), and variable name \(x\)
- Apply \(f\) to the declared type to get type of \(x\)
  \[
  \text{Vardec}(\text{TypA}(\text{TypP}(\text{TypI}), \text{Some 4}), \text{"x"})
  \]
Interpreting micro-C

• Interpreter data:
  – locEnv, *environment* mapping local variable names to store addresses
  – gloEnv, *environment* mapping global variable names to store addresses, and global function names to (parameter list, body statement)
  – *store*, mapping addresses to (integer) values

• Main interpreter functions:
  exec: stmt -> locEnv -> gloEnv -> store -> store
  eval: expr -> locEnv -> gloEnv -> store -> int * store
  access: access -> locEnv -> gloEnv -> store -> address * store
Micro-C statement execution

- As for the naïve language, but two envs:

```plaintext
let rec exec stmt locEnv gloEnv store : store =
    match stmt with
    | If(e, stmt1, stmt2) ->
        let (v, store1) = eval e locEnv gloEnv store
        if v<>0 then exec stmt1 locEnv gloEnv store1
        else exec stmt2 locEnv gloEnv store1
    | While(e, body) ->
        let rec loop store1 =
            let (v, store2) = eval e locEnv gloEnv store1
            if v<>0 then loop (exec body locEnv gloEnv store2)
            else store2
        loop store
    | ...  
```
Expression statements in C, C++, Java and C#

• The “assignment statement”
  \[ x = 2+y; \]
  is really an expression
  \[ x = 2+y \]
  followed by a semicolon

  Value: none
  Effect: change \( x \)

  Value: 2+y
  Effect: change \( x \)

• The semicolon means: ignore value

let rec exec stmt locEnv gloEnv store : store =
  match stmt with
  | ... |
  | Expr e ->
    let (_, store1) = eval e locEnv gloEnv store
    store1

Evaluate expression then ignore its value
Micro-C expression evaluation, 1

- Evaluation of an expression
  - takes local and global env and a store
  - gives a resulting *rvalue* and a *new store*

```ocaml
and eval e locEnv gloEnv store : int * store =
  match e with
  | ...|
  | CstI i       -> (i, store)
  | Prim2(ope, e1, e2) ->
    let (i1, store1) = eval e1 locEnv gloEnv store
    let (i2, store2) = eval e2 locEnv gloEnv store1
    let res =
      match ope with
      | "*"   -> i1 * i2
      | "+"   -> i1 + i2
      | ...   -> i1 + i2
    (res, store2)
```
Micro-C expression evaluation, 2

- To evaluate access expression `x, *p, arr[i]`
  - find its lvalue, as an address `loc`
  - look up the rvalue in the store, as `store1[loc]`

- To evaluate `&e`
  - just evaluate `e` as lvalue
  - return the lvalue

```plaintext
and eval e locEnv gloEnv store : int * store =
match e with
| Access acc ->
  let (loc, store1) = access acc locEnv gloEnv store
  (getSto store1 loc, store1)
| Addr acc -> access acc locEnv gloEnv store
| ...  
```
Micro-C access evaluation, to \textit{lvalue}

- A variable $x$ is looked up in environment
- A dereferencing $*e$ just evaluates $e$ to an address
- An array indexing $\text{arr}[\text{idx}]$
  - evaluates $\text{arr}$ to address $a$, then gets $\text{aval}=\text{store}[a]$
  - evaluates $e$ to rvalue index $i$
  - returns address $(\text{aval}+i)$

```
and access acc locEnv gloEnv store : int * store =
  match acc with
  | AccVar x             -> (lookup (fst locEnv) x, store)
  | AccDeref e           -> eval e locEnv gloEnv store
  | AccIndex(acc, idx)   ->
    let (a, store1) = access acc locEnv gloEnv store
    let aval = getSto store1 a
    let (i, store2) = eval idx locEnv gloEnv store1
    (aval + i, store2)
```
Operators &x and *p are inverses

- The address-of operator &e
  - evaluates e to its lvalue
  - returns the lvalue (address) as if it were an rvalue

- The dereferencing operator *e
  - evaluates e to its rvalue
  - returns the rvalue as if it were an lvalue

- It follows
  - that &(*e) equals e
  - that *(&e) equals e, provided e has lvalue
Micro-C, interpreter and compiler

• This lecture: Interpretation of micro-C

Micro-C program → run in Interp.fs → Output

ex1.c

• Next lecture: Compilation of micro-C

Micro-C program → Compiler → Machine code → Machine.java → Output

ex1.c → ex1.out
Reading and homework

• This week’s lecture:
  – PLCSD chapter 7
  – Strachey: Fundamental Concepts ...
  – Kernighan & Ritchie: The C programming language, chapter 5.1-5.5

• Next lecture
  – PLCSD chapter 8