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

Peter Sestoft
Monday 2013-09-30
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

- Abstract syntax
- 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)
- 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;
```

```
sum = 0;
while sum < 10000 do begin
    sum = sum + i;
    i = 1 + i;
end;
```
Naïve-store statement execution, 1

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

```haskell
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:

\[
\begin{align*}
\text{exec } (\text{Block } [s_1; s_2]) \text{ store} &= \text{loop } [s_1; s_2] \text{ store} \\
&= \text{exec } s_2 \ (\text{exec } s_1 \text{ store})
\end{align*}
\]

```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)
    loop store
```
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

```
i:  42
sum: 44
```

<table>
<thead>
<tr>
<th></th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>...</td>
<td></td>
<td></td>
<td>5050</td>
<td></td>
</tr>
</tbody>
</table>
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#

```
p:  42
sum: 44
```

- 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 $tmp$
Possible evaluation of main(3)

main(3):
    fac(3, 117):
      fac(2, 118):
        fac(1, 119):
          fac(0, 120):
            *120 = 1
            *119 = 1 * 1
            *118 = 1 * 2
            *117 = 2 * 3
          print 6
Storage model for micro-C

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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>globals</td>
<td>main</td>
<td>fac(3)</td>
<td>fac(2)</td>
<td>fac(1)</td>
</tr>
</tbody>
</table>

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

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;
}
```

Addresses:
- `a`: 41
- `b`: 42
- `x`: 43
- `y`: 44
- `tmp`: 45

Store:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>22</td>
<td>22</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
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 may help:

cdecl> explain int *(*ipap)[4]  
declare ipap as pointer to array 4 of pointer to int  
cdecl> declare n as array 7 of pointer to pointer to int  
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` : \( \text{TypI} \)
  - `int *x` : \( \text{TypP}(\text{TypI}) \)
  - `int x[4]` : \( \text{TypA}(\text{TypI}, \text{Some 4}) \)

- **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) { ... }`
```plaintext
type typ =
| TypI                             (* Type int                    *)
| TypC                             (* Type char                   *)
| TypA of typ * int option         (* Array type                  *)
| TypP of typ                      (* Pointer type                *)

and expr =
| 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 =
| AccVar of string                 (* Variable access        x    *)
| AccDeref of expr                 (* Pointer dereferencing  *p   *)
| AccIndex of access * expr        (* Array indexing         a[e]   *)

and stmt =
| If of expr * stmt * stmt         (* Conditional                 *)
| While of expr * stmt             (* While loop                  *)
| Expr of expr                     (* Expression statement e;    *)
| Return of expr option            (* Return from method          *)
| Block of stmtordec list          (* Block: grouping and scope   *)

and stmtordec =
| Dec of typ * string              (* Local variable declaration *)
| Stmt of stmt                      (* A statement                 *)

and topdec =
| Fundec of typ option * string * (typ * string) list * stmt
| Vardec of typ * string

and program =
| Prog of topdec list
```

Lexer specification for micro-C

- New: endline comments  
  and delimited comments

```plaintext
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:
  \[
  \text{int} \,*x[4]
  \]

- 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:

```ocaml
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

• The semicolon means: ignore value

```ocaml
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*

```plaintext
and eval e locEnv gloEnv store : int * store =
  match e with
    | ...               -> (i, store)
    | CstI i            -> (i, store)
    | Prim2(o, e1, e2)  ->
      let (i1, store1) = eval e1 locEnv gloEnv store
      let (i2, store2) = eval e2 locEnv gloEnv store1
      let res =
        match o with
          | "*"   -> i1 * i2
          | "+"   -> i1 + i2
          | ...   -> i1 + i2
      (res, store2)
```
To evaluate access expression $x, *p, arr[i]$
- find its lvalue, as an address $loc$
- look up the rvalue in the store, as $store_1[loc]$

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 $(\text{aval} + i)$

```plaintext
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

• Next lecture: Compilation of micro-C
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