Thu 11/7/2013: Runtime environments

Lecture topics:

I. Overview
II. Stack
III. Helper functions
IV. Heap

I. Overview

Compiler → Assembly code (.s) → Assembly → Relocatable machine code (.o)

Libraries (runtime.o) → Linker/Loader → Target machine code (a.out)

High addresses

Stack

<table>
<thead>
<tr>
<th>Free memory</th>
<th>Heap</th>
<th>Static</th>
<th>Code</th>
</tr>
</thead>
</table>

Function activation records (aka activation frames)

records, arrays, strings

string literals

instructions

Low addresses

II. Stack

main = (f(); g();)
f = if ... then f() else g()
g = (...) 

Example Tigu:

k := f(Ei, ..., En); 

function f(x1, ..., x_n)
    ...

Example Tigu IR:

E1, code: 

E2, code: 

Function returns last expression E in body 

E.code: return E.addr;

Stack frame layout (see x64-intra.pdf)

<table>
<thead>
<tr>
<th>Caller's frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actuals</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>Caller's %rbp</td>
</tr>
<tr>
<td>Locals/ temporaries</td>
</tr>
</tbody>
</table>

Base pointer (%rbp) → Address of callee's instruction after call

Call sequence on x64:

- push actuals in registers/stack} caller
- push %rbp
- move %rbp, %esp
- sub %esp, frame size

Return sequence on x64:

- move %rax, return value
- move %r8, %rbp
- pop %rbp
- ret (pops return address)
- remove actuals from stack} callee
- use %rax

Other stack topics:

- saving registers across calls
- variable-size data on stack
- accessing non-local stack variables

III. Helper functions

Intrinsics: see Tigu-spec.pdf

- print, flush, getchar,
- ord, chr, size, substring, concat,
- not, exit

Support for complex IR instructions; see ir-spec

- if addr1, relبن, addr2 goto label
- relبن := 1 <= | <= 1 <= |
- complex if type is string!
- addr1 := new [addr2]
- allocate array of addr1 elements
- addr1 := new
- allocate record of type of addr1

Write all these functions in C,

use gcc to compile them to runtime.o,

then link them against a generated by compile
IV  Heap

let
  type R = {a : int}

var
  x :: R {a = 1}
  y :: R {a = 2}

in
  x :: nil;

y
end

Lifetimes of heap objects are not nested
and may be indefinite

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Java</th>
<th>C</th>
<th>C++</th>
<th>Tiger</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>new</td>
<td>malloc</td>
<td>new</td>
<td>allocate</td>
</tr>
<tr>
<td></td>
<td>garbage</td>
<td>free</td>
<td>delete</td>
<td>unspecified</td>
</tr>
</tbody>
</table>

Example Tiger

```c

 MareX, y :: int;
p :: R, x :: MareX, y :: MareY;
```

Example Tiger IR

```
allocate
```

Mark-sweep garbage collection (GC):
- Traverse objects from roots to find reachable
- Reclaim all unreachables
- Use mark stack (reached, must scan points)
- Mark bit in each heap objects

States of object:
```
Free
```

```
Allocate
```

```
Unreachable
```

- Not on mark stack
- Mark bit is 0

```
Reach
```

```
Scan
```

```
Scanned
```

- Not on mark stack
- Mark bit is 1

```
Unscanned
```

- On mark stack
- Mark bit is 1

```
Mark Stack
```

(overflow)
```
A
```

```
B
```

```
D
```

```
E
```

```c

Example:

void* nr(int size)
{
  void* result;
  result = find OnFree List (size);
  if (result != 0)
  {
    return result;
  }
  else
  {
    garbage collection C;
    result = find OnFree List (size);
    if (result != 0)
    {
      return result;
    }
    else
    {
      printf (stderr, "Out of memory");
      exit (-1);
    }
  }
}
```