CS143: Runtime

David L. Dill
Stanford University
Runtime

• Semantic Analysis (wrap up)
  • Error Recovery

• Runtime Conventions
  • Memory Regions
  • Stack Allocation
  • Expression Evaluation on a Stack
Error Recovery
Goal: Report errors after the first one

let \( y : \text{Int} \leftarrow x + 2 \) in \( y + 3 \)

Type for undeclared \( x \)?

If \( x : \text{Object} \), \( x + 2 \) will cause another error.

Reports too many errors.
Another Solution

Compiler stores "No-type" for erroneous expressions

No-type ≤ C for all types C
so No-type ⊔ C is always C

All operations/assignments are ok.

let y: Int ← x + 2 in y + 3 ← only one error

\[ \text{No-type} \sqcup \text{Int} \rightarrow \text{Int} \]
No-type

Types no longer a tree

Not a problem unless you have code that assumes it's a tree

Not required in project
Runtime Conventions
Memory Regions
Machine architecture does not (usually) dictate how:

- Memory is used
- Function calls work
  etc.

But these need to be handled consistently.

E.g., Function calls: caller & callee need to agree on where arguments will be.
Memory Organization

Process has many different memory regions
Set up by OS, initialization code.
I will ignore dynamic loading, shared memory, etc.
Memory Regions

Code - the instructions

Data - several kinds
  Global variables & constants
  Stack - for function call data
  Heap - for other dynamically created data (e.g. from "malloc" in C)

Executable code needs to be able to find these.
Finding Data

Global Variables and constants

Occupy fixed addresses in memory
Compiler decides where they go.

Heap-allocated objects

Pointers are stored/passed in global, local variables, parameters, etc.

Formals & Locals: Next
Stack Allocation
Use of Stack
Designed for sequential code
Breaks down if
In multithreaded code.
If functions can access variables
of containing functions
AND functions are treated as first-class
values (requires "closures")
Assume these are not issues for this lecture.
Program Stack

During function call, need temporary storage for:

- Actual parameter values
- Local variables
- Saved machine state (e.g., registers)
- Return address
- Return value
- Pointer to previous activation record
  (sometimes other stuff)

↑ "Control link"
Function Calls

call \ f_1
   call \ f_2
      call \ f_3
         return from \ f_3
         return from \ f_2
         return from \ f_1
Function Calls

```
[ call f1
  call f2
    call f3
      return from f3
      return from f2
    return from f1
  return from f2
]
```

Nesting structure will work.
Def: Data to execute a function is stored in an activation record (aka "stack frame").

Under above assumptions, these can be stored in the stack.

Actual parameter values
Local variables
Saved machine state (e.g. registers)
Return address
Return value
Control link
(sometimes other stuff)
Stacks Grow Down (for now)
A R organization

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>arguments</td>
</tr>
<tr>
<td>control link</td>
</tr>
<tr>
<td>return address</td>
</tr>
</tbody>
</table>

There are many possibilities
Compiler writer must decide and generate code to
set up / tear down
access data

Responsibilities must be divided between callee and callee.

Design Considerations:

- Simplicity of codegen
- Efficiency (minimize data moves)
- Compatibility with other code.
Class Main {
    g(): Int { 13 }
    f(x: Int): Int { if x == 0 then g() else f(x-1) }
    main(): Int { f(3); 33; }  
}

result
argument
control link
return address

Main

(current frame)

return address

f

(result+)
3
(*)
Class Main {
    g(c) : Int \{ 1, 3 \};
    f(x : Int+) : Int \{ if \ x == 0 \ then \ g(c) \ else \ f(x-1) \};
    main() : Int \{ \{ f(3); \}; \};
}
\(*\)

```
result
argument
control link
return address
```

```
Main
\( f \)
\( (result) \)
\( 3 \)
\( (\ast) \)

AR for main is not useful
```

return address
Class Main {
  g(): Int { 1 3 }
  f(x: Int): Int { if x = 0 then g() else f(x-1) } /*
  main(): Int { f(3); } */ /*
}

result
argument
control link
return address

Main
  f
  (result+)
  3
  (*)
  (result+)
  2
  (**)
Class Main {
    g(): Int { 1 3; }
    f(x: Int+): Int { if x = 0 then g() else f(x-1); }
    main(): Int { f(3); }
}
Heap Allocation

Problem: Some data lives longer than the function call that creates it.
Solution: Heap allocation.

Heap: One or more reserved memory regions

C: malloc — marks free memory as used.
   free — marks used memory as free

Many languages: New A allocates a new instance of A

Storage may be reclaimed automatically (e.g. garbage collection)
Memory Alignment

Almost all modern architectures are byte addressable (8-bit bytes)

But many instructions require (or strongly reward) alignment of data to 32-bit or 64-bit boundaries.

E.g., A 32-bit integer at address 10007 would not work well with an add instruction

"Padding" - unused bytes so next object is aligned.
Expression Evaluation with a Stack
Intermediate Values

Evaluate left-to-right bottom-up.

Need to save a * b, c + d, while computing e - f.
Intermediate Values

Evaluate left-to-right bottom-up.
Need to save \(a \times b\), \(c + d\), while computing \(e - f\).
This requires temporary storage,
Intermediate Values

Stack (grows up)

```
2 3 * + 4 5 6 7
```
Intermediate Values

```
  +  
  / 
  *  *
  |  |
  2 3 +--
     /   |
    4 5 6 7  
```

Stack (grows up)

```
3 2
```
Intermediate Values

Stack (grows up)
Intermediate Values

Stack (grows up)

4
6
Intermediate Values

Stack (grows up)

5
4
6
Intermediate Values

Stack (grows up)

9
6
Intermediate Values

\[
\begin{array}{c}
+ \\
+ \\
* \\
\ \ 2 3 \\
\ \ 4 5 6 7
\end{array}
\]

Stack (grows up)

6
9
6
Intermediate Values

```
        +
       /|
      / \
    2    3
   / \   /\  \
  4   5  6   7
```

Stack (grows up)

```
7  6  9  6
```
Intermediate Values

Stack (grows up)

-1
9
6
Intermediate Values

Stack (grows up)

-9
6
Intermediate Values

\[
\begin{align*}
&\text{Stack (grows up)} \\
&-3
\end{align*}
\]
Code Generation for Expressions

Instructions:
push i - push the number i on the stack
add - pop top two numbers, add them, push the sum.

2 + 3 \implies \text{push 2} \\
\text{push 3} \\
\text{add}
Code Generation for Stack Machines is Easy

No need to optimize scarce registers

Don't need to specify operands always the top n values on the stack.

No need to specify destinations

Short instructions & compact programs.
Stack + Accumulator

Idea: Top of stack is heavily used keep it in a fast register (called the "accumulator" - acc)

add: acc ← acc + top of stack (does not pop top of stack)