CS 536 Announcements for Tuesday, April 12, 2022

Last Time
- type checking
- type-system concepts
- type-system vocabulary
- minim
  - type rules
  - how to apply type rules

Today
- runtime environments
- runtime storage layout
- activation records
- static allocation
- stack allocation
- activation records revisited
- what happens on function call, entry, return

Next Time
- parameter passing

Type checking in minim

minim's type system
- primitive types: int bool void
- type constructors: struct
- coercion: a bool cannot be used as an int is expected and vice versa

Type errors in minim
- operators applied to operands of wrong type
- expressions that, because of context, must be a particular type but are not
- related to function calls

Type checking
- Recursively walks the AST to
  - determine the type of each expression and sub-expression using the type rules of
    the language
  - find type errors
- Add a typeCheck method to AST nodes
**Type checking: errors**

**Goals:**
- report as many *distinct* errors as possible.
- don't report *same* error multiple times – avoid error cascading

Introduce internal **error** type
- when type incompatibility is discovered
  - report the error
  - pass error up the tree
- when a type check gets **error** as an operand
  - don't (re)report an error
  - pass error up the tree

**Example:**
```c
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```
Back to the big picture

Before code generation, we need to consider the runtime environment:

- underlying software & hardware configuration assumed by the program

Program piggybacks on the operating system (OS)

- provides functions to access hardware
- enforces some boundaries on what is allowed

Compiler must use runtime environment as best it can

- limited # of very fast registers to do computation
- comparatively large region of memory to hold data
- some basic instructions from which to build more complex behaviors

We need to create/impose conventions on the way our program accesses memory

- assembly code enforces very few rules
- conventions help to guarantee separately developed code works together

Issues to consider

Variables

- How are they stored?
- What happens when a variable's value is needed?

How do functions work?

- What information should be stored for each function?
- What should happen when client code calls a function?
- What should happen when a function is entered?
- What should happened when a function returns?
Storage layout

Region for global memory
- code, static data

One "frame" for each procedure/function
- memory "slot" for each local, parameter
- memory "slot" for caller

- every time a function is called, its names (local vars & params) refer to same loc in memory
- fast access to all names
- no overhead of stack/heap manipulations
- no recursion
- no dynamic data structures
Memory layout: stack allocation

Allocate one activation record (AR) per invocation
- use the stack
- push a new AR on function entry
- pop AR on function exit
- to reduce the size, put static data in the global area

Stack size not known at compile time
- don't know (at compile-time) how many ARs there will be
- size of local variables may not be known
- each AR keeps track of:
  - its boundaries
  - the previous AR's boundaries

Activation record keeps track of
- local variables
- info about the call made by the caller
  - data context
    - enough info to determine boundaries of AR in use when current function was called
  - control context
    - enough info to know what code invoked the current function

Non-local dynamic memory

Don't always want all data allocated in a function call to disappear on return
- want to be able to create and link lists

Don't know how much space we'll need
- allocate many such objects of varying sizes

The Heap
- region of memory independent of the stack
- allocated according to calls in the program
- how is memory "given back"?
  - programmer specifies when no longer in use (C)
  - runtime environment determines automatically when no longer in use (Java)
Function calls

Instruction pointer ($ip$) tracks the line (address) of code that it is executing

- if $ip$ points to code generated for some function, we'll say we are in that function

**caller** = function doing the invocation

**callee** = function being invoked

\[ \text{$sp$} - stack pointer, \text{points to top of stack (1st unused location)} \]

\[ \text{$fp$} - frame pointer, \text{points to bottom of current AR} \]

Activation records revisited

<table>
<thead>
<tr>
<th>low address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>local variables</td>
</tr>
<tr>
<td>saved registers</td>
</tr>
<tr>
<td>control link</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>parameters</td>
</tr>
<tr>
<td>return value</td>
</tr>
</tbody>
</table>

\[ \text{for non-void function (or might return via special register)} \]

\[ \text{used to pop AR from stack when from returns} \]

\[ \text{stack growth} \]
Function entry: **caller** responsibilities

- Store the *caller-saved* registers in it's own AR
- Set up the actual parameters
  - set aside slot for the return value
  - push parameters onto the stack
- Copy return address out of $sp
- Jump to first instruction of the callee

Function entry: **callee** responsibilities

- Save $fp (it will need to be restored when the callee returns)
- Update the base of the new AR to be the end of the old AR
- Save *callee-saved* registers (if necessary)
- Make space for locals

$$sp = sp - (\text{size of local data})$$
$$fp = sp + \text{size of control link}$$
$$fp = sp + \text{size of ret addr}$$
$$fp = sp + \text{size of param}$$

Function exit: **callee** responsibilities

- Set the return value
- Restore callee-saved registers
- Grab stored return address
- Restore *old* $sp
- Restore *old* $fp
- Jump to the stored return address

Function exit: **caller** responsibilities

- Pop the return value (or copy from register)
- Restore caller-saved registers
Example

```c
#1 int summation(int max) {
#2    int sum = 1;
#3    for (int k = 1; k <= max; k++) {
#4        sum += k;
#5    }
#6    return sum;
#7 }
#8 void main() {
#9    int x = summation(4);
#10   print(x);
#11 }
```