CS 536 Announcements for Monday, April 1, 2024

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

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

Next Time
- parameter passing

Type checking in base

base's type system
- primitive types: integer logical void
- type constructors: tuple
- coercion: a logical cannot be used as an integer is expected and vice versa

Type errors in base
- 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:
```
integer a.
logical 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 access to hardware
- provides illusion of uniqueness
- 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

- allows modularity
- increases programmer efficiency

**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?
General memory layout

Region for global memory

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

Every time a function is called, its names (local variables & parameters) refer to the same location in memory

+ Fast access to all names
+ No overhead for stack manipulations
- No recursion
- No dynamic memory allocation
  (eg, linked lists)
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 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 curr func was called
  - control context
    - enough info to know code that invoked curr func

Non-local dynamic memory

Don't always want all data allocated in a function call to disappear on return

Don't know how much space we'll need
- allocate many such objs of various 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

$sp$ (stack pointer) – points to top of stack

$fp$ (frame pointer) – points to bottom of current AR

Activation records revisited
**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

---

**Function exit: callee responsibilities**

Set the return value

Restore callee-saved registers

Grab stored return address

Restore *old* $sp from control link

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
integer summation(integer max) {
    integer sum.
    integer k.
    sum = 0.
    k = 1.
    while k <= max {
        sum = sum + k.
        k++.
    }
    return sum.
}

void main() {
    integer x.
    x = summation(4).
    write << x.
}
```