CS 412/413

Introduction to
Compilers and Translators

Andrew Myers
Cornell University

Lecture 34: Optimizing first-class functions
24 April 00

Administration

- Programming Assignment 5 due today
- Programming Assignment 6 design report due next Friday (5^{th})
- Reading: Appel 15.3-15.6

First-class functions

- No lexical nesting (C)
  - fast but limited
  - function value is pointer to code
- Lexical nesting, no upward function values or storage in data structures (Pascal, Modula-*n*):
  - function value is *closure*

code
environment

- Fully first-class: return values (F_2, Scheme, ML):
  - lots of heap-allocation, more indirection
  - Functions roughly as powerful as objects (sometimes more convenient), but as expensive as objects… without optimization

Objects via records and 1st-class functions

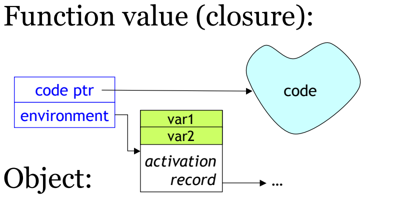
class Foo {
  f_1: T_1 … f_n: T_n
  Foo(a_1, …, a_n) = e_c
  m_1() = e_1
  …
  m_n() = e_n
}

Foo(a_1, …, a_n) = (
  f_1: T_1, …, f_n: T_n;
  e_c;
  return new record {
    m_1 = (function() = e_1),
    …
    m_n= (function() = e_n)
  }
)

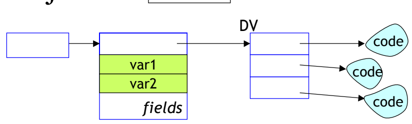
- Object is record of closures for every method
- Doesn't handle:
  - references to this in e_i
  - inheritance

Functions vs. objects

- Function value (closure):



- Object:



Inefficiencies

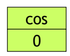
- Functions more expensive to call (extra environment argument to pass to code)
- Slow access to local variables (on heap)
- Slower access to non-local variables (chaining through activation records)
- Activation records heap allocated—much more garbage to collect
- Closure values keep all lexically containing activation records reachable—hard to collect garbage
- *How to have cake, eat it too?*

Top-level calls

- Top-level functions: no lexical environment; no implicit static link argument needed

float cos(x: float)

- Call directly to code; don't pass environment pointer: cos(x) → `call _cos`
- Form closure to top-level fcn with dummy environment



  - calls to cos via closure *will* pass dummy env.
  - make last (optional) argument, pass in dedicated register, or have two entry points (`_cos`, `_cos_closure`) with diff. calling conventions

Calls to nested fcns

- Evaluation of expression g can be optimized if used for function call: don't construct closure explicitly

need closure here

```
int f(n: int,
      g1: function(): int) = (
  g(): int = n;
  if (n == 0) f(1, g, dummy)
  else g1() + g()
)
```

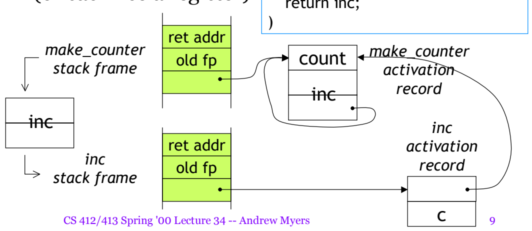
just need code address, fp *here*

CALL(NAME(g), FP, ...)

Local variable access

- Activation record holds local vars
- Can't store on stack
- Indirection needed! (or sacrifice a register)

```
make_counter( ): function( ): int = (
   // returns a counter function
   int count;
   inc( ): int = ( c: int; count++; )
   return inc;
)
```



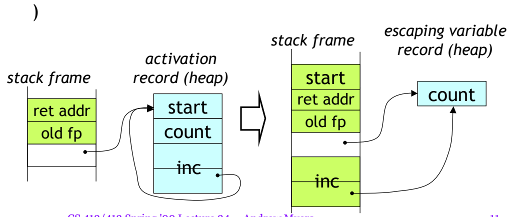
Escape analysis

- Idea: local variable only needs to be stored on heap if it can *escape* and be accessed after this function returns
- Only happens if
  - variable is referenced from within some nested function
  - the nested function is turned into a closure:
    - returned, or
    - passed to some function that might store it in a data structure

    (calls to nested functions not a problem)
- This determination: *escape analysis*

Example

```
make_counter(start: int): function( ): int = (
   // returns a counter function
   int count = start;
   inc( ): int = ( c: int; count++; )
   return inc;
)
```



Benefits of escape analysis

- Variables that don't escape are allocated on stack frame instead of heap: cheap to access
- If no escaping variables, no heap allocation at all (common case)
- Closures don't pin down as much garbage when created
- One problem: precise escape analysis is a global analysis, expensive. Escape analysis must be conservative

Limitations of escape analysis

```
interface set { members(f: function(o: object)) }
countAnimals(s: set) = (
        count: int = 0;
        loop_body(o: object) = (
                if (cast(o, Animal)) count ++;
        );
        s.members(loop_body);
        return count;
)
```

Does count escape?

Escape analysis for objects

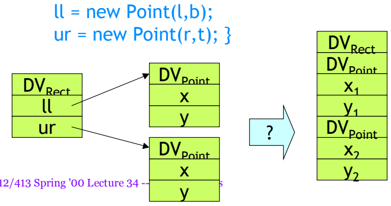
- Objects ≈ functions …
- Can use escape analysis to allow stack-allocation of some objects
  - if created by new `C()` in current func.
  - & reference to object never stored in
    - global variable
    - another *heap*-allocated object
    - & never passed to a function that might store it
  - Fast alloc/dealloc, fast access
- Rare optimization: doesn't work that often! (methods calls unsafe; need global analysis)

(C++: can specify heap or stack, but unsafe)

Inlining object layouts

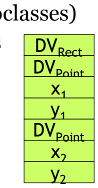
- More effective optimization: inlining object fields

```
class Rectangle {
    private: Point ll, ur;
    public: Rectangle(int l,r,t,b) {
        ll = new Point(l,b);
        ur = new Point(r,t); }
}
```



Conditions

- Effect: less memory fragmentation, fewer indirections, faster code
- Can inline object fields if
  - field always initialized in constructor to same known class (*e.g.* Point)
  - field never assigned otherwise (need enscapsulation even against subclasses)
  - GC can handle internal pointers to subobjects
  - specialization can help
- Current research!



Summary

- How to get back to the performance of a language with 2nd class functions:
  - call top-level functions w/o static link argument
  - don't construct closures on calls
  - use escape analysis to avoid heap-allocating most variables
- Escape analysis ideas apply to optimization of objects too