CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 35: First-class functions
27 Apr 01

Administration

- Programming Assignment 6 write-up due in one week
  - register allocation
  - constant folding
  - unreachable code elimination
- Reading: Appel 15.1-15.6

Advanced Language Support

- "advanced" language features so far: objects
- Next four lectures: more modern language features
  - first-class functions
  - exceptions
  - parametric polymorphism
  - dynamic typing and meta-object protocols

First-class vs. Second-class

- Values are first-class if they can be used in all the usual ways
  - assigned to local variables
  - passed as arguments to functions/methods
  - returned from functions
  - created at run-time
- Iota: modules, functions are denoted by expressions but are only usable in limited ways (uses, function call)

First-class functions

- Many languages allow functions to be used in a more first-class manner than in Iota or Java: C, C++, ML, Modula-3, Pascal, Scheme,...
  - Passed as arguments to functions/methods
  - Nested within containing functions (exc. C, C++)
  - Used as return values (exc. Modula-3, Pascal)

Function Types

- Iota-F_0: Iota$^+$ with function values that can be passed as arguments (still not fully first-class)
- Need to declare type of argument; will use program notation function(T_1, T_2): T_3 to denote the function type $T_1 \times T_2 \rightarrow T_3$.
- Example: sorting with a user-specified ordering:

```
sort(a: array[int],
    order: function(int, int):bool) {
  … if (order(a[i], a[j])) { … } …
```

Passing a Function Value

```
leq(x: int, y: int): bool = x <= y
geq(x: int, y:int): bool = x >= y
sort(a: array[int],
     order: function(int, int):bool) …

  sort(a1, leq)
  sort(a2, geq)
```

- Allows abstraction over choice of functions

Objects subsume functions!

```
interface comparer {
  compare(x: int, y:int): bool
}
sort(a: array[int], cmp: comparer) {
  … if (cmp.compare(a[i], a[j])) { … } …

class leq implements comparer {
  compare(x: int, y:int) = x <= y;
}
sort(a1, new leq);
```


Type-checking functions

- Same rules as in Iota static semantics, but function invoked in function call may be a general expression

$$\frac{f : T_1 \times \ldots \times T_n \rightarrow T_R \in A \qquad A \vdash e_i : T_i \quad i \in 1..n}{A \vdash f(e_1, \ldots, e_n) : T_R} \Rightarrow \frac{A \vdash e_0 : T_1 \times \ldots \times T_n \rightarrow T_R \qquad A \vdash e_i : T_i \quad i \in 1..n}{A \vdash e_0(e_1, \ldots, e_n) : T_R}$$

- Subtyping on function types: usual contravariant/covariant conditions

Representing functions

- For Iota-F_0, a function may be represented as a pointer to its code (cheaper than an object)
- *Old translation*:
  $\mathcal{E}[\![f(e_1,\ldots,e_n)]\!] =$ CALL(NAME(f), $\mathcal{E}[\![e_1]\!],\ldots,\mathcal{E}[\![e_n]\!]$)
- *New*: $\mathcal{E}[\![e_0(e_1,\ldots,e_n)]\!] =$ CALL($\mathcal{E}[\![e_0]\!], \mathcal{E}[\![e_1]\!],\ldots,\mathcal{E}[\![e_n]\!]$)
  $\mathcal{E}[\![id]\!]$ = NAME(id)
  (if id is a global fcn)

Nested Functions

- In functional languages (Scheme, ML) and Pascal, Modula-3, Iota-F_1
- Nested function can access variables of the containing *lexical* scope

```
plot_graph(f: function(x: float): float)=
      ( … y = f(x) … )
plot_quadratic(a,b,c: float) = (
  q(x: float): float = a*x*x+b*x+c;
  plot_graph(q)
)
```

nested function *free variables*

Iteration in Iota-F_1

- Also useful for iterators, other user-defined control flow constructs

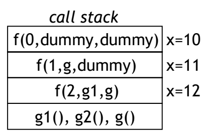
```
interface set { members(f: function(o: object)) }
countAnimals(s: set) = (
  count: int = 0;
  loop_body(o: object) = (
     if (cast(o, Animal)) count ++;
  )
  s.members(loop_body);
  return count;
)
```

- Nested functions may access, update free variables from containing scopes! Must change function representation

A subtle program

```
int f(n: int,
      g1: function(): int,
      g2: function(): int) = (
  int x = n+10;
  g(): int = x;
  if (n == 0) f(1, g, dummy)
  else if (n==1) f(2, g1, g)
  else g1() + g2() + g()
)
```

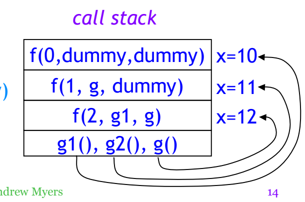
f(0,dummy,dummy) = ?



Lexical scope

- g(): int = x creates a *new function value*
- Free variable (x) is bound to the variable *lexically visible* at evaluation of function expression

```
int f(n: int,
      g1: function(): int,
      g2: function(): int) = (
  int x = n+10;
  g(): int = x;
  if (n == 0) f(1, g, dummy)
  else if (n==1) f(2, g1, g)
  else g1() + g2() + g()
)
```

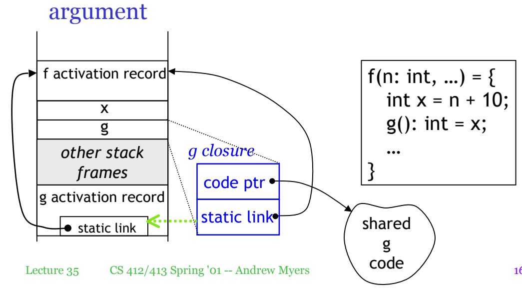


Closures

- Problem: nested function (g) may need to access variables *arbitrarily* high up on stack
- Before nested functions: function value was pointer to code (1 word)
- With nested functions: function value is a *closure* of code + environment for free variables (2 words)

Closure

- *Closure* -- A pointer to the code **plus** a *static link* to allow access to outer scope
- Static link passed to function code as implicit argument

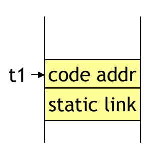


Supporting Closures

$\mathcal{E}[\![e_0(e_1,\ldots,e_n)]\!]$ =
  ESEQ(MOVE(t1, $\mathcal{E}[\![e_0]\!]$),
  CALL(MEM(t1), MEM(t1+4), $\mathcal{E}[\![e_1]\!]$,…, $\mathcal{E}[\![e_n]\!]$)

implicit static link argument

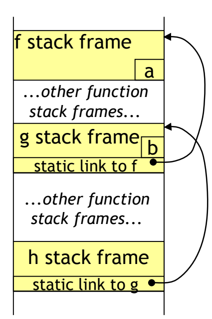
$\mathcal{S}[\![id(..a_i{:}\ T_i..) : T_R = e]\!]$=
  t1 = FP − k_{id};
  [t1] = NAME(*id*);
  [t1+4] = FP;



- Can optimize direct calls
- Function variable takes 2 stack locations
- What about variable accesses?

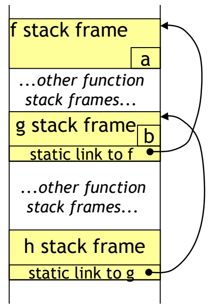
Static Link Chains

```
f() = (a: int;
  g() = (b:int;
    h() = (
      c = a + b;
    ) ...
  ) ...
)
```



Variable access code

- Local variable access unchanged
- Free variable access: walk up *n* static links before indexing to variable



Progress Report

√ Passed as arguments to functions/methods

√ Nested within containing functions as local variables

—Used as return values

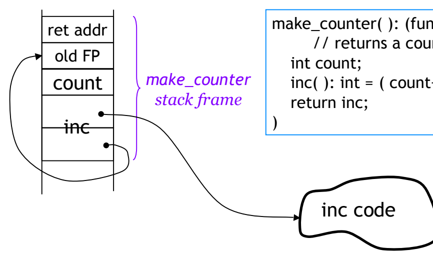
- If no nested functions, functions are just pointers to code; can be used as return values (C)
- Problem: interaction with nested fcns

Iota-F_2 (first-class functions)

- Augment Iota-F_1 to allow the return type of a function to be a function itself.

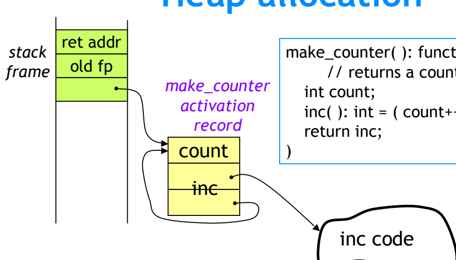
```
make_counter( ): (function( ): int) = (
      // returns a new counter function
  int count = 0;
  inc(): int = ( count++ );
  return inc
)
make_counter()() + make_counter()() = ?
c = make_counter(); c() + c() + c() = ?
```


Dangling static link!




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


Heap allocation




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

- Solution: heap-allocate the make_counter activation record (at least count)
- Activation record ≠ stack frame
- Even *local* variable accesses indirected

The GC side-effect

- Every function call creates an object that must be garbage collected eventually -- increases rate of garbage generation
- Activation records of all lexically enclosing functions are reachable from a closure via stack link chains
- Activation record makes a lot of garbage look reachable!

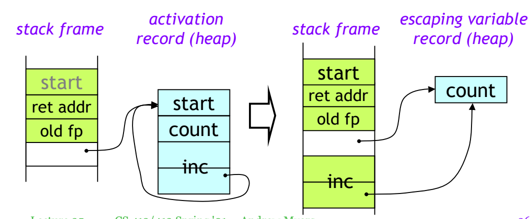
Escape analysis

- 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 *and*
  - 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.

Summary

- Looked at 3 languages progressively making functions more first-class
- No lexical nesting (F_0, C)
  - Fast but limited
  - Function = pointer to code
- Lexical nesting, no upward function values or storage in data structures (F_1, Pascal, Modula-[123]):
  - function value is *closure*
- 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