



# COMP 181

---


Lecture 20  
*Compiling functional languages*

December 5, 2005

## Class status


- **Project 2:** grading is going a bit slowly
- **Project 4:** linear scan register allocation (Due Dec 12 – during the reading period)
- **Final:** take-home, due at the time the final would be
- **Today:** course evaluations



2


## Prelude

- Who is this guy?  
Alonzo Church
- What is he famous for?
  - *Church's Thesis* (also Church-Turing Thesis)  
What is that?
    - Any calculation that is computable has an algorithm
  - Undecidability  
First paper to show that a problem is undecidable. (Predates Turing's work on halting problem)
  - Lambda calculus
    - Formulated for undecidability work
    - Needed a simple, universal notion of a computation



**Doctoral students**

Alan Turing  
Stephen Kleene  
J. Rosser  
Michael O. Rabin




3

## Lemma

Sam's list of "Biggest Bummers in Math/CS"

- P probably not equal to NP
- Undecidability
  - There are questions for which there is no *effective* way to compute the answer
- Godel's incompleteness theorem
  - Any logic system of sufficient power cannot be both *consistent* and *complete*
- Russell's paradox
  - What is "the set of all sets that do not contain themselves"?



4


## Functional languages

- What is a functional language?

**Informally:** *a language in which functions are taken seriously*

**Formally:** *a language that supports the manipulation of functions as first-class values, in the style of the  $\lambda$ -calculus*

- Examples
  - ML (Caml, OCaml), Lisp, Scheme, Haskell, Miranda




5

## $\lambda$ -calculus

- Invented by Alonzo Church and Stephen Kleene
  - Characterize all computable functions
  - Simple language
  - Computations are entirely mechanical
- This had not been done before
- How simple?
 

```

expr  →  identifier
        |  λ identifier . expr
        |  ( expr expr )
      
```



6

## Informally

- Language of anonymous functions
  - $\lambda x. x * 2$
  - Takes an argument  $x$  and multiplies by 2
  - Note:
    - Special symbols "\*" and "2"
    - We can construct these from the basic grammar
- Variables
  - Bound** variable: occurs in expression qualified by  $\lambda$
  - Free** variable: not mentioned in any  $\lambda$

$$\lambda x. x * n$$

Tufts University Computer Science 7

## Informally

- How do we make functions with more than one argument?
  - Key:** functions can return other functions
- $\lambda n. \lambda x. x * n$ 
  - Multiplies two arguments
  - Outer function takes  $n$ , and returns a function that multiplies by  $n$
- $\lambda n x. x * n$
- Currying**
  - Fix one argument

Called **Higher-order functions**

$$(\lambda n x. x * n) 2 = \lambda x. x * 2$$

Tufts University Computer Science 8

## $\lambda$ -calculus

Two semantic operations:

- $\alpha$ -conversion
  - Informally: the variable names don't matter

$$\lambda n. n \text{ and } \lambda x. x$$

- $\beta$ -reduction
  - Informally: apply a function to an argument
  - Key: it's just symbol substitution

$$(\lambda x. e) y = e[x := y]$$

- Note: must be careful with free variables (use  $\alpha$ -conversion)

Tufts University Computer Science 9

## Primitives

- How can we define numbers?
  - $0 = \lambda f x. x$
  - $1 = \lambda f x. f x$
  - $2 = \lambda f x. f (f x)$
  - $3 = \lambda f x. f (f (f x))$
- A "number" is a function that takes a single arg function and returns a single arg function
- The result consists of  $N$  applications of input function
- Yes, it will blow your mind

Tufts University Computer Science 10

## Primitives

- Successor function
  - Takes a number  $n$  and returns  $n+1$
  - More precisely: the functions representing  $n$  and  $n+1$

$$\text{SUCC} = \lambda n f x. f (n f x)$$

Lambda expression for 1

$$\text{SUCC } 1 = \lambda n f x. f (n f x) (\lambda f x. f x)$$

$$\lambda f x. f ((\lambda f x. f x) f x)$$

$$\lambda f x. f f x$$

Lambda expression for 2

- Others:
  - $\text{ADD} = \lambda m n f x. m f (n f x)$
  - $\text{MULT} = \lambda m n f. m (n f)$

Tufts University Computer Science 11

## Strictness

- Order of evaluation
- Given a function application, either:
  - Strict:** Evaluate the arguments first, then pass into function
  - Non-strict:** Pass in the arguments, evaluate only as necessary
- Examples:
  - Strict: ML, Scheme, lisp
  - Non-strict: Haskell, Miranda
- What's the advantage of non-strict language?
  - May never need to evaluate the arguments
  - Arguments may be impossible to evaluate!

**primes:** a function to generate all prime numbers **first (primes)**

Tufts University Computer Science 12

## Big ideas

- Very simple model of computation
- Computations are entirely mechanical – symbol pushing
- No “state”
- Making it practical:
  - Provide the primitives automatically
  - Provide some data structures (Lisp lists, s-expressions)
  - What about performance?



## Execution

Three execution models

- *Interpretation* *BASIC, perl, etc.*  
Control (the sequence of computations) is represented by a giant tree-shaped expression (a term). The interpreter walks this tree at run-time and applies  $\beta$ -reductions
- *Native compilation* *C, Fortran*  
Control is compiled down to machine instructions and executed directly on the hardware. Ends up like a specialization of the interpreter code for the given expression.
- *Compilation to an abstract machine* *Java, C#*  
Control is compiled down to instructions in an abstract machine. The abstract machine is designed to more closely match the semantics of the language



## Issues in compilation

Consider “scale” function:

```
let scale =  $\lambda n x. x * n$ 
```

```
let scale_2 = scale 2 =  $\lambda x. x * 2$ 
```

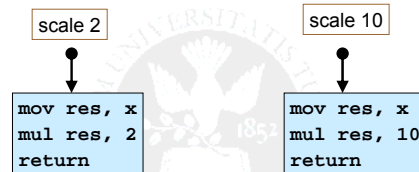
```
let scale_10 = scale 10 =  $\lambda x. x * 10$ 
```

- What does a compiled function look like?
  - Get input arguments from stack or register
  - Compute on arguments
  - Leave result on stack or in register
- What’s the problem with this?  
*Higher order functions should generate compiled code*



## Scale

```
let scale =  $\lambda n x. x * n$ 
```



- Problem:
  - Requires run-time code generation
  - Every higher-order function has a compiler in it



## Better solution

Notice:

```
let scale =  $\lambda n x. x * n$ 
```

The code generated by a call to scale (with only one argument) always has the same basic structure:

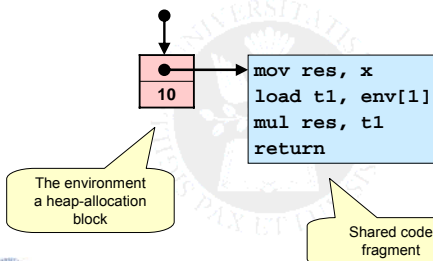
```
mov res, x
mul res, <value of n>
return
```

- Idea:
  - Share common code – like a code template
  - Put parts that vary (free variables) in a common structure called the *environment*



## For scale

Return value of scale 10



## Closures

- This data structure is called a *closure*
  - Closures are dynamically allocated
  - Contain two things
    - A **code pointer** – points to a fixed piece of compiled code
    - An **environment** – a record providing plug-in values for the free variables.
- To execute:
  - Move the environment pointer into a common register
  - Call the code piece
- Key to fast interpreters and compilers for functional languages



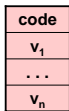
## Representing closures

- Depends on how much is known about the function returning the closure and the application of the closure
- Most limiting*: nothing is known about a closure
  - Any function could be called
  - The code part must be in a fixed position in the closure
- More flexible*: the environment is only accessed when the closure is constructed and when the code is executed
  - Two sides can agree on a structure
  - May be able to store parts of environment into registers

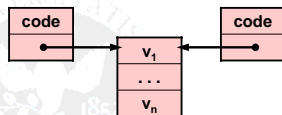


## Examples

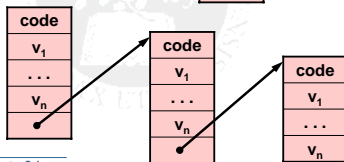
One-block closure



Two-block closures (with environment sharing)



Linked closures



## Trade-offs

- Time trade-off
  - One-block closures:
    - Slower to build
    - Faster to access variables.
  - Linked closures:
    - Faster to build – chain together nested calls
    - Slow to access – must follow the chains
- Space trade-off:
  - Minimal environments:
    - Bind only the free variables
    - Fewer opportunities to share environments
  - Larger environments
    - Bind additional variables
    - May cause space leaks



**Modern compilers**



## Recursive functions

- Recursive functions need access to their own closure
- Several options:
  - Reconstruct the closure on the fly, using the current environment
  - Treat the function as a free variable in its own environment – results in a cycle in the closures
  - One-block: reuse the same input closure



## Abstract machines

- Example: expressions
  - Compile to stack machine
  - Convert nested expressions to postfix notation
    - add x y becomes x y +
    - $C(op(a_1, \dots, a_n))$  becomes  $C(a_1); \dots; C(a_n); op;$
- Machine
  - A "program counter" indexing the stream of ops
  - A stack with intermediate results
- Execution
  - Look at the next op, increment program counter
  - Pop some args of stack, apply op
  - Push result on stack
  - Done: consumed all ops – result is left on stack



## Abstract machine for strict language

- Three components
  - a program counter
  - a stack for intermediate results and return frames
  - an environment giving values to free variables
- Compilation scheme
  - $C(n)$  becomes ACCESS( $n$ )
  - $C(\lambda.a)$  becomes CLOSURE( $C(a)$ ); RETURN
  - $C(a\ b)$  becomes  $C(a)$ ;  $C(b)$ ; APPLY
- Expression as in the previous example



## Execution

- ACCESS( $n$ )
  - Look up variable  $n$  in environment  $e$ , push on stack
- CLOSURE( $c'$ )
  - Push pair ( $c', e$ ) on stack
- APPLY
  - Pop argument  $v$  and the closure to apply ( $c', e'$ )
  - Add  $v$  to environment  $e'$
  - Save return state – push ( $c, e$ ) on stack
  - Start executing at  $c'$
- RETURN
  - Restore frame – pop ( $c, e$ ) off stack
  - Leave return value on top of stack



## Making it fast

- Abstract machine interpreter
  - Written in C
  - Store closures as arrays of arguments, arrays of abstract machine operations
  - Giant switch over ACCESS, CLOSURE, APPLY, etc.
  - An order of magnitude faster than term-level interpreter
- Compile down
  - Replace abstract machine ops with real machine ops
  - Code fragment for each op
  - Need to make sure registers are used properly
- Trickier for non-strict languages



## Advantages

- "Compilation" is a systematic process of rewriting terms
- Abstract machine has well-defined semantics
- What does that suggest?
  - *Can prove correctness of the translation*
- This has been done for some compilers
  - More general notion: *proof-carrying code*
  - Code being compiled carries a proof that the translation is correct.



## Compilation to machine code

- **Step 1:**
  - Replace functions by closures
  - Make explicit the construction, passing, access of environments
- Yields an intermediate language
  - Similar to conventional intermediate languages
  - Manipulates code pointers (closed functions)
- **Step 2:**
  - Optimize and generate code from intermediate language
  - Write your own: OCaml, SML/NJ
  - Use a C compiler: GHC, Bigloo



## From functions to closures

- Every function:
  - Assume args are stored in environment array
  - Generate code that loads values and computes on them
- At a function call:
  - Load the environment
  - Get the code pointer and jump to the address
- What's the problem with that scheme?
  - *Every call is an indirect, computed jump*
  - Bad branch prediction, stalls pipeline
  - Typically 10x or so more expensive than static call



## Static calls

- Many calls can be statically resolved

```
let succ = λx. x + 1 in succ (succ 2)
let rec f = λx. ... f arg ...
```

- Conditions
  - Application within scope of function definition
  - Recursive calls
  - Higher-order functions applied only once
- What can we do?
  - Generate calls to statically known addresses
  - Inline the code – what's the “functional” name for this?  
β-reduction



## Control-flow analysis

- Different notion of control
- Idea:  
Approximate the set of functions that could be called at each function application point
  - Track which functions end up at which call sites
  - Gives an approximation of the *call graph*
    - Nodes are functions
    - Edges are calls
- If that set is a singleton
  - Generate direct call or inline



## CFA

- Remember: functions are first-class values
  - We need to track the flow of functions
  - CFA is a dataflow analysis
- Typically set up as a constraint system
  - “the functions that reach this site are a subset of the functions that reach some earlier point”
  - Solving the basic algorithm is  $O(n^3)$
- Applications
  - Optimize function calls in functional languages
  - Also: optimize virtual dispatch in OO languages
  - Eliminate run-time tests in languages like Java, Scheme



## Next time

- Last lecture
  - A bit more on function languages – data representations
  - Wrap –up
- Take home final
  - Won't be as bad as the mid-term
  - Will be tested on someone else

