COS320: Compiling Techniques

Zak Kincaid

April 13, 2022

Compiling functional languages

Functional languages

- First class functions: functions are values just like any other
  - can be passed as parameters (e.g., `map`)
  - can be returned (e.g. `(+) 1`)

Functional languages

- First class functions: functions are values just like any other
  - can be passed as parameters (e.g., `map`)
  - can be returned (e.g. `(+) 1`)
- Functions that take functions as parameters or return functions are called *higher-order*

Functional languages

- First class functions: functions are values just like any other
  - can be passed as parameters (e.g., `map`)
  - can be returned (e.g. `(+) 1`)
- Functions that take functions as parameters or return functions are called *higher-order*
- A higher-order functional language is one with *nested functions* with *lexical scope*

Scoping

- $(\textbf{fun}\ x\ \texttt{->}\ e)$ is an expression that evaluates to a function
  - x is the function's parameter
  - e is the function's body
- Occurrences of x within e are said to be *bound* in $(\textbf{fun}\ x\ \texttt{->}\ e)$
  - Variables are resolved to most closely containing **fun**.
- Occurrences of variables that are not bound are called *free*

$$(\textbf{fun}\ x\ \texttt{->}\ (\textbf{fun}\ y\ \texttt{->}\ (x\ z)\ (\textbf{fun}\ x\ \texttt{->}\ x)\ y))$$

Closures

- Consider $((\textbf{fun}\ x\ \texttt{->}\ (\textbf{fun}\ y\ \texttt{->}\ x))\ 0)\ 1$
  1. Apply the function $(\textbf{fun}\ x\ \texttt{->}\ \textbf{fun}\ y\ \texttt{->}\ x)$ to the argument 0 $\rightsquigarrow$ $(\textbf{fun}\ y\ \texttt{->}\ x)$

Closures

- Consider $((\textbf{fun}\ x\ \texttt{->}\ (\textbf{fun}\ y\ \texttt{->}\ x))\ 0)\ 1$
  1. Apply the function $(\textbf{fun}\ x\ \texttt{->}\ \textbf{fun}\ y\ \texttt{->}\ x)$ to the argument 0 $\rightsquigarrow (\textbf{fun}\ y\ \texttt{->}\ x)$
  2. Apply the function $(\textbf{fun}\ y\ \texttt{->}\ x)$ to the argument 1 $\rightsquigarrow ???$
     - x is free in $(\textbf{fun}\ y\ \texttt{->}\ x)$!

Closures

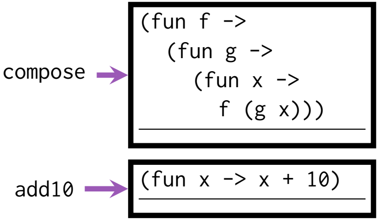
- Consider $((\textbf{fun } x \texttt{ -> } (\textbf{fun } y \texttt{ -> } x))\ 0)\ 1$
  1. Apply the function $(\textbf{fun } x \texttt{ -> } \textbf{fun } y \texttt{ -> } x)$ to the argument 0 $\rightsquigarrow (\textbf{fun } y \texttt{ -> } x)$
  2. Apply the function $(\textbf{fun } y \texttt{ -> } x)$ to the argument 1 $\rightsquigarrow ???$
     - x is free in $(\textbf{fun } y \texttt{ -> } x)$!
- In higher-order functional languages, a function value is a *closure*, which consists of a function pointer *and* an environment
  - Environment is used to interpret variables from enclosing scope

```
let compose =
  fun (f : int -> int) ->
    (fun (g : int -> int) ->
      (fun (x : int) ->
        f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
```

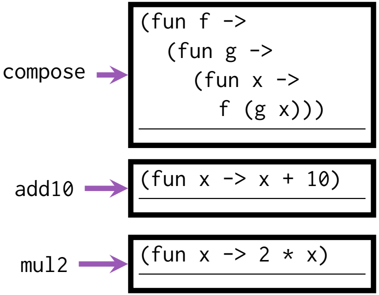
compose

```
(fun f ->
  (fun g ->
    (fun x ->
      f (g x)))
```

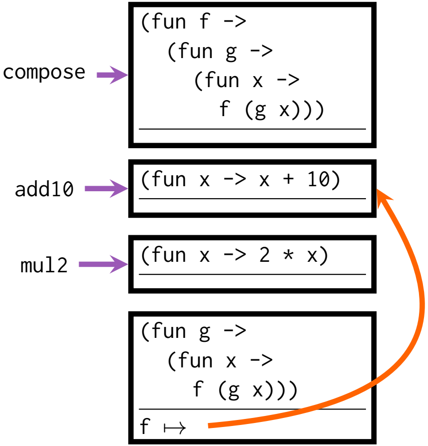
```
let compose =
  fun (f : int -> int) ->
    (fun (g : int -> int) ->
      (fun (x : int) ->
        f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
```



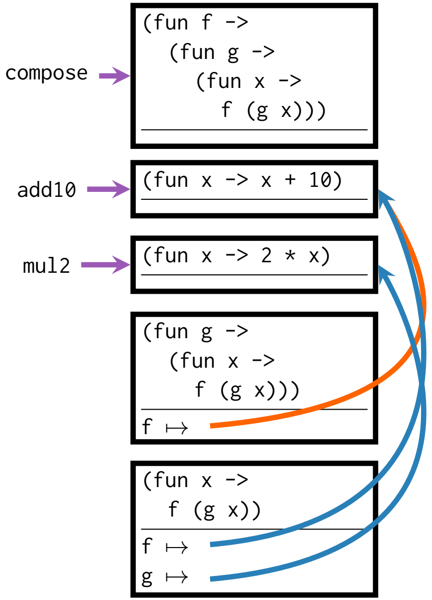
```
let compose =
  fun (f : int -> int) ->
    (fun (g : int -> int) ->
      (fun (x : int) ->
        f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
```



```
let compose =
  fun (f : int -> int) ->
    (fun (g : int -> int) ->
      (fun (x : int) ->
        f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
```



```
let compose =
  fun (f : int -> int) ->
    (fun (g : int -> int) ->
      (fun (x : int) ->
        f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
```



Compiling closures

- Strategy: translate a language with closures to one with (just) function pointers
- *Closure conversion* transforms a program so that no function accesses free variables
- *Hoisting* transforms a closure-converted program so that all function expressions appear at the top-level
  - Function expressions can be implemented as functions

Nameless representation

- Idea (de Bruijn): use a representation of expressions without named bound variables
  - Each variable is replaced by a number: # of enclosing scopes between occurrence & the scope it is resolved to
  - $($**fun** x `->` $x) \rightsquigarrow ($**fn** $0)$
  - $($**fun** x `->` $($**fun** y `->` $x)) \rightsquigarrow ($**fn**$($**fn** $1))$

Nameless representation

- Idea (de Bruijn): use a representation of expressions without named bound variables
  - Each variable is replaced by a number: # of enclosing scopes between occurrence & the scope it is resolved to
  - (**fun** x `->` x) $\rightsquigarrow$ (**fn** 0)
  - (**fun** x `->` (**fun** y `->` x)) $\rightsquigarrow$ (**fn**(**fn** 1))
  - (**fun** x `->` (**fun** y `->` y)) $\rightsquigarrow$

Nameless representation

- Idea (de Bruijn): use a representation of expressions without named bound variables
  - Each variable is replaced by a number: # of enclosing scopes between occurrence & the scope it is resolved to
  - (**fun** x `->` x) $\leadsto$ (**fn** 0)
  - (**fun** x `->` (**fun** y `->` x)) $\leadsto$ (**fn**(**fn** 1))
  - (**fun** x `->` (**fun** y `->` y)) $\leadsto$(**fn**(**fn** 0))

Nameless representation

- Idea (de Bruijn): use a representation of expressions without named bound variables
  - Each variable is replaced by a number: # of enclosing scopes between occurrence & the scope it is resolved to
  - (**fun** x `->` x) $\leadsto$ (**fn** 0)
  - (**fun** x `->` (**fun** y `->` x)) $\leadsto$ (**fn**(**fn** 1))
  - (**fun** x `->` (**fun** y `->` y)) $\leadsto$(**fn**(**fn** 0))
  - (**fun** x `->` (**fun** y `->` x) x) $\leadsto$

Nameless representation

- Idea (de Bruijn): use a representation of expressions without named bound variables
  - Each variable is replaced by a number: # of enclosing scopes between occurrence & the scope it is resolved to
  - (**fun** x `->` x) $\leadsto$ (**fn** 0)
  - (**fun** x `->` (**fun** y `->` x)) $\leadsto$ (**fn**(**fn** 1))
  - (**fun** x `->` (**fun** y `->` y)) $\leadsto$(**fn**(**fn** 0))
  - (**fun** x `->` (**fun** y `->` x) x) $\leadsto$(**fn**(**fn** 1) 0)

Nameless representation

- Idea (de Bruijn): use a representation of expressions without named bound variables
  - Each variable is replaced by a number: # of enclosing scopes between occurrence & the scope it is resolved to
  - (**fun** x `->` x) $\rightsquigarrow$ (**fn** 0)
  - (**fun** x `->` (**fun** y `->` x)) $\rightsquigarrow$ (**fn**(**fn** 1))
  - (**fun** x `->` (**fun** y `->` y)) $\rightsquigarrow$(**fn**(**fn** 0))
  - (**fun** x `->` (**fun** y `->` x) x) $\rightsquigarrow$(**fn**(**fn** 1) 0)
- Environments can be implemented as lists
  - Each environment has a pointer to parent environment

Closure conversion

- Invariant: translated expressions involve a single variable, say p
  - p represents an *environment* (as a list)
- Variable x (with index i) $\rightsquigarrow$ look-up ith element of p

Closure conversion

- Invariant: translated expressions involve a single variable, say p
  - p represents an *environment* (as a list)
- Variable x (with index i) $\rightsquigarrow$ look-up ith element of p

$$(\textbf{fun}\ x\ \texttt{->}\ e) \rightsquigarrow (\textbf{fun}\ p\ \texttt{->}\ e', p) \qquad \text{where } e \rightsquigarrow e'$$

$$(f\ a) \rightsquigarrow (\textbf{fst}\ f')\ (a' :: (\textbf{snd}\ f')) \qquad \text{where } f \rightsquigarrow f',\ a \rightsquigarrow a'$$

Closure conversion

- Invariant: translated expressions involve a single variable, say p
  - p represents an *environment* (as a li
- Variable x (with index i) $\leadsto$ look-up ith element of

Save evaluation environment

$$(\textbf{fun}\ x\ \texttt{->}\ e) \leadsto (\textbf{fun}\ p\ \texttt{->}\ e', p) \qquad \text{where } e \leadsto e'$$

$$(f\ a) \leadsto (\textbf{fst}\ f')\ (a' {::} (\textbf{snd}\ f')) \qquad \text{where } f \leadsto f',\ a \leadsto a'$$

Closure conversion

- Invariant: translated expressions involve a single variable, say p
  - p represents an *environment* (as a li

Save evaluation environment

- Variable x (with index i) $\leadsto$ look-up ith element o

$$(\textbf{fun}\ x\ \texttt{->}\ e) \leadsto (\textbf{fun}\ p\ \texttt{->}\ e', p) \qquad \text{where } e \leadsto e'$$

$$(f\ a) \leadsto (\textbf{fst}\ f')\ (a' :: (\textbf{snd}\ f')) \qquad \text{where } f \leadsto f',\ a \leadsto a'$$

Evaluation environment: index $0 \mapsto a$, other indices shifted

Practical closure conversion

- Following a chain of pointers for each variable access is expensive

Practical closure conversion

- Following a chain of pointers for each variable access is expensive
- Partially flattened representation: environment is represented as a list of arrays
    - List stores bindings for entire *activation frames* rather than single variables

Practical closure conversion

- Following a chain of pointers for each variable access is expensive
- Partially flattened representation: environment is represented as a list of arrays
  - List stores bindings for entire *activation frames* rather than single variables
- Flattened representation: environment is represented as an array
  - Fast accesses
  - Greater space requirement (no sharing with parent environment)

Practical closure conversion

- Following a chain of pointers for each variable access is expensive
- Partially flattened representation: environment is represented as a list of arrays
    - List stores bindings for entire *activation frames* rather than single variables
- Flattened representation: environment is represented as an array
    - Fast accesses
    - Greater space requirement (no sharing with parent environment)
    - Can reduce space by storing only variables that are actually free

Hoisting

- After closure-conversion, every function expression is closed (no free variables)
  - No free variables $\Rightarrow$ no need for closures
  - Function expressions simply evaluate to (C-style) function pointers

Hoisting

- After closure-conversion, every function expression is closed (no free variables)
  - No free variables $\Rightarrow$ no need for closures
  - Function expressions simply evaluate to (C-style) function pointers
- *Hoisting*:
  - Gives globally unique identifiers each function expression
  - Replaces function expressions with their identifiers
  - Places definitions for the identifiers as top-level scope

Functional optimizations

- Tail call elimination: functional languages favor recursion over loops, but loops are more efficient (need to allocate stack frame, push return address, save registers, ...)
  - Tail call elimination searches for the pattern

    ```
    %x = call foo ...; ret %x
    ```

    and compiles the `call` as a jump instead of a `callq`

Functional optimizations

- **Tail call elimination**: functional languages favor recursion over loops, but loops are more efficient (need to allocate stack frame, push return address, save registers, …)
  - Tail call elimination searches for the pattern

    ```
    %x = call foo ...; ret %x
    ```

    and compiles the `call` as a jump instead of a `callq`
- **Function inlining**: functional programs tend to have lots of small functions, which incurs the cost of more function calls than there may be in an imperative language
  - *Inlining* replaces function calls with their definitions to alleviate some of this burden

Functional optimizations

- Tail call elimination: functional languages favor recursion over loops, but loops are more efficient (need to allocate stack frame, push return address, save registers, ...)
  - Tail call elimination searches for the pattern

    ```
    %x = call foo ...; ret %x
    ```

    and compiles the `call` as a jump instead of a `callq`
- Function inlining: functional programs tend to have lots of small functions, which incurs the cost of more function calls than there may be in an imperative language
  - *Inlining* replaces function calls with their definitions to alleviate some of this burden
- Uncurrying: in some functional languages (e.g., OCaml), functions always take a single argument at a time
  - E.g., in `let f x y = ...`, `f` takes one argument `x`, and returns a closure which takes a second argument `y` and produces the result
  - A single OCaml-level function call may result in *several* function calls and closure allocations
  - *Uncurrying* is an optimization that determines when a function is always called with more that one parameter `(f 3 4)`, and compiles it as a multi-parameter function.