

COS320: Compiling Techniques

Zak Kincaid

January 29, 2026

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

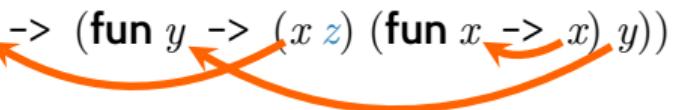
- First class functions: functions are values just like any other
 - can be passed as parameters (e.g., `map`)
 - can be returned (e.g. `(+) 1`)
- Functions that take functions as parameters or return functions are called *higher-order*

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

- $(\text{fun } x \rightarrow e)$ 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 $(\text{fun } x \rightarrow e)$
 - Variables are resolved to most closely containing **fun**.
- Occurrences of variables that are not bound are called *free*

$$(\text{fun } x \rightarrow (\text{fun } y \rightarrow (x \text{ } z) (\text{fun } x \rightarrow x) \text{ } y))$$


Closures

- Consider $((\text{fun } x \rightarrow (\text{fun } y \rightarrow x)) 0) 1$
 - ➊ Apply the function $(\text{fun } x \rightarrow \text{fun } y \rightarrow x)$ to the argument $0 \rightsquigarrow (\text{fun } y \rightarrow x)$

Closures

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

Closures

- Consider $((\text{fun } x \rightarrow (\text{fun } y \rightarrow x)) 0) 1$
 - ① Apply the function $(\text{fun } x \rightarrow \text{fun } y \rightarrow x)$ to the argument $0 \rightsquigarrow (\text{fun } y \rightarrow x)$
 - ② Apply the function $(\text{fun } y \rightarrow x)$ to the argument $1 \rightsquigarrow ???$
 - x is free in $(\text{fun } y \rightarrow x)$!
- 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
```

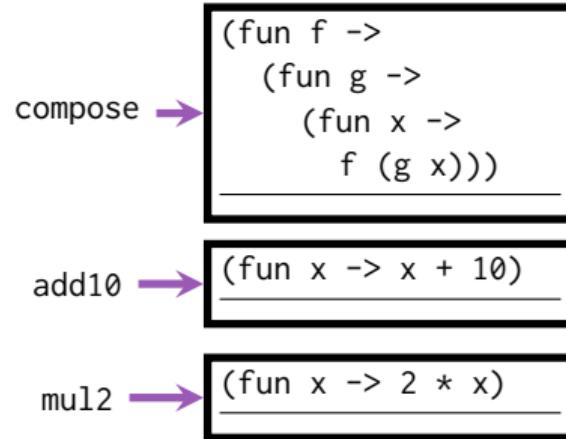
compose →

$$\begin{array}{l} (\text{fun } f \rightarrow \\ \quad (\text{fun } g \rightarrow \\ \quad \quad (\text{fun } x \rightarrow \\ \quad \quad \quad f (g x)))) \end{array}$$

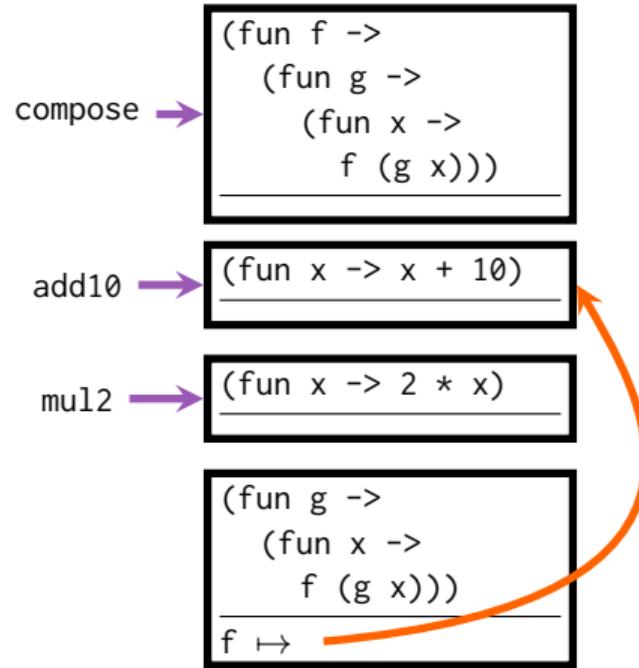
add10 →

$$\begin{array}{l} (\text{fun } x \rightarrow x + 10) \end{array}$$

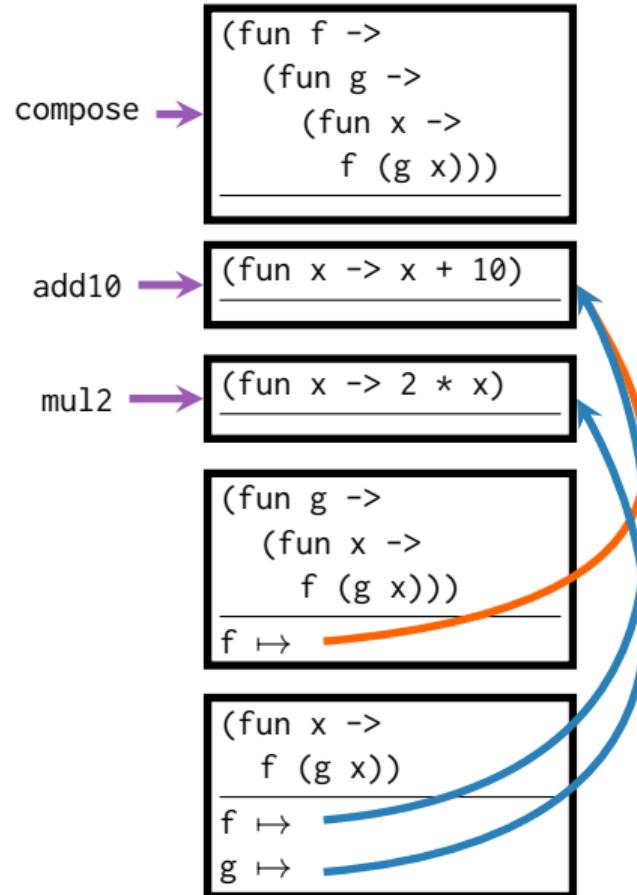
```
let compose =
  fun (f : int -> int) ->
    (fun (g : int -> int) ->
      (fun (x : int) ->
        f (g x)))
let add10 = fun (x : int) -> x + 10
let mul2 = fun (x : int) -> 2 * x
let result = compose add10 mul2 100
```



```
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
 - $(\mathbf{fun} \ x \ -> \ x) \rightsquigarrow (\mathbf{fn} \ 0)$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ x)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 1))$

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
 - $(\mathbf{fun} \ x \ -> \ x) \rightsquigarrow (\mathbf{fn} \ 0)$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ x)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 1))$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ y)) \rightsquigarrow$

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
 - $(\mathbf{fun} \ x \ -> \ x) \rightsquigarrow (\mathbf{fn} \ 0)$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ x)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 1))$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ y)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 0))$

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
 - $(\mathbf{fun} \ x \ -> \ x) \rightsquigarrow (\mathbf{fn} \ 0)$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ x)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 1))$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ y)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 0))$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ x) \ x) \rightsquigarrow$

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
 - $(\mathbf{fun} \ x \ -> \ x) \rightsquigarrow (\mathbf{fn} \ 0)$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ x)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 1))$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ y)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 0))$
 - $(\mathbf{fun} \ x \ -> \ (\mathbf{fun} \ y \ -> \ x) \ x) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 1) \ 0)$

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
 - $(\mathbf{fun} \ x \ -\!> \ x) \rightsquigarrow (\mathbf{fn} \ 0)$
 - $(\mathbf{fun} \ x \ -\!> \ (\mathbf{fun} \ y \ -\!> \ x)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 1))$
 - $(\mathbf{fun} \ x \ -\!> \ (\mathbf{fun} \ y \ -\!> \ y)) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 0))$
 - $(\mathbf{fun} \ x \ -\!> \ (\mathbf{fun} \ y \ -\!> \ x) \ x) \rightsquigarrow (\mathbf{fn}(\mathbf{fn} \ 1) \ 0)$
- 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 i th 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 i th element of p

$$\begin{array}{ll} (\mathbf{fun} \ x \rightarrow \ e) \rightsquigarrow (\mathbf{fun} \ p \rightarrow \ e', p) & \text{where } e \rightsquigarrow e' \\ & \text{where } f \rightsquigarrow f', a \rightsquigarrow a' \\ \downarrow & \downarrow \\ (f \ a) \rightsquigarrow (\mathbf{fst} \ f) \ (a' :: (\mathbf{snd} \ f)) & \end{array}$$

Closure conversion

- Invariant: translated expressions involve a single variable, say p

- p represents an *environment* (as a list)

Save evaluation environment

- Variable x (with index i) \rightsquigarrow look-up i th element of p

(fun x \rightarrow e) \rightsquigarrow **(fun** p \rightarrow e', p)

where $e \rightsquigarrow e'$

\downarrow
(f a **)** \rightsquigarrow **(****fst** f **)** $(a' ::$ **(****snd** f **)****)**

where $f \rightsquigarrow f'$, $a \rightsquigarrow a'$

Closure conversion

- Invariant: translated expressions involve a single variable, say p

- p represents an *environment* (as a list)

Save evaluation environment

- Variable x (with index i) \rightsquigarrow look-up i th element of p

$(\mathbf{fun} \ x \rightarrow \ e) \rightsquigarrow (\mathbf{fun} \ p \rightarrow \ e', p)$ where $e \rightsquigarrow e'$

$(f \ a) \rightsquigarrow (\mathbf{fst} \ f) \ (a' :: (\mathbf{snd} \ f))$ where $f \rightsquigarrow f$, $a \rightsquigarrow a'$

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 than one parameter (`f 3 4`), and compiles it as a multi-parameter function.