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 \(\text{fun}\).
- Occurrences of variables that are not bound are called *free*

\[(\text{fun } x \rightarrow (\text{fun } y \rightarrow (x z) (\text{fun } x \rightarrow x) y))\]
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 \(0 \rightarrow (\text{fun } y \rightarrow x)\)

Environment is used to interpret variables from enclosing scope

In higher-order functional languages, a function value is a \textit{closure}, which consists of a \textit{function pointer} and an environment.
Closures

- Consider \((\text{fun } x \to (\text{fun } y \to x))\) 0 1
  1. Apply the function \((\text{fun } x \to \text{fun } y \to x)\) to the argument 0 \(\mapsto (\text{fun } y \to x)\)
  2. Apply the function \((\text{fun } y \to x)\) to the argument 1 \(\mapsto ???\)
    - \(x\) is free in \((\text{fun } y \to x)!\)
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 \(0 \rightarrow (\text{fun } y \rightarrow x)\)
  2. Apply the function \((\text{fun } y \rightarrow x)\) to the argument \(1 \rightarrow ???\)
     - \(x\) 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
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
  - \((\text{fun} \ x \rightarrow \ x) \leadsto (\text{fn} \ 0)\)
  - \((\text{fun} \ x \rightarrow (\text{fun} \ y \rightarrow x)) \leadsto (\text{fn}(\text{fn} \ 1))\)

Environments can be implemented as lists  
Each environment has a pointer to parent environment
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
  - $(\text{fun}\ x \to\ x) \leadsto (\text{fn}\ 0)$
  - $(\text{fun}\ x \to (\text{fun}\ y \to\ x)) \leadsto (\text{fn}(\text{fn}\ 1))$
  - $(\text{fun}\ x \to (\text{fun}\ y \to\ y)) \leadsto$
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
  - $(\text{fun } x \rightarrow x) \mapsto (\text{fn } 0)$
  - $(\text{fun } x \rightarrow (\text{fun } y \rightarrow x)) \mapsto (\text{fn(fn } 1))$
  - $(\text{fun } x \rightarrow (\text{fun } y \rightarrow y)) \mapsto (\text{fn(fn } 0))$

Environments can be implemented as lists
Each environment has a pointer to parent environment
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
  - $\text{(fun } x \rightarrow x) \rightarrow (\text{fn } 0)$
  - $\text{(fun } x \rightarrow (\text{fun } y \rightarrow x)) \rightarrow (\text{fn(fn 1)})$
  - $\text{(fun } x \rightarrow (\text{fun } y \rightarrow y)) \rightarrow (\text{fn(fn 0)})$
  - $\text{(fun } x \rightarrow (\text{fun } y \rightarrow x) x) \rightarrow$
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
  - \((\text{fun } x \rightarrow x) \rightarrow (\text{fn } 0)\)
  - \((\text{fun } x \rightarrow (\text{fun } y \rightarrow x)) \rightarrow (\text{fn } (\text{fn } 1))\)
  - \((\text{fun } x \rightarrow (\text{fun } y \rightarrow y)) \rightarrow (\text{fn } (\text{fn } 0))\)
  - \((\text{fun } x \rightarrow (\text{fun } y \rightarrow x) x) \rightarrow (\text{fn } (\text{fn } 1) 0)\)

Environments can be implemented as lists
Each environment has a pointer to parent environment
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
- \((\text{fun}\ x \rightarrow x) \rightsquigarrow (\text{fn}\ 0)\)
- \((\text{fun}\ x \rightarrow (\text{fun}\ y \rightarrow x)) \rightsquigarrow (\text{fn}(\text{fn}\ 1))\)
- \((\text{fun}\ x \rightarrow (\text{fun}\ y \rightarrow y)) \rightsquigarrow (\text{fn}(\text{fn}\ 0))\)
- \((\text{fun}\ x \rightarrow (\text{fun}\ y \rightarrow x)\ x) \rightsquigarrow (\text{fn}(\text{fn}\ 1)\ 0)\)

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 $i$) $\leadsto$ look-up $i$th element of $p$

\[
(\text{fun } x \rightarrow e) \leadsto (\text{fun } p \rightarrow e', p)
\]

\[
(f \ a) \leadsto (\text{fst } f') (a'::(\text{snd } f'))
\]

where $e \leadsto e'$

where $f \leadsto f'$, $a \leadsto a'$
Closure conversion

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

\[
(f u n \ x \ \rightarrow \ e) \rightsuccsim (f u n \ p \ \rightarrow \ e', \ p) \quad \text{where } e \rightsuccsim e'
\]

\[
(f \ a) \rightsuccsim (f s t \ f') (a'::(s n d \ f')) \quad \text{where } f \rightsuccsim f', \ a \rightsuccsim a'
\]
Closure conversion

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

$(\text{fun } x \rightarrow e) \leadsto (\text{fun } p \rightarrow e', p)$ where $e \leadsto e'$

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

- Save evaluation environment
- 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
    \[
    \%x = \text{call } \text{foo} \ldots; \text{ret } \%x
    \]
  - 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
    \[
    \%x = \text{call} \text{foo} \ldots; \text{ret} \%x
    \]
    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
    \[
    \%x = \text{call foo} \ldots; \text{ret} \ %x
    \]
    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 \texttt{let f x y = \ldots}, \texttt{f} takes one argument \texttt{x}, and returns a closure which takes a second argument \texttt{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 (\texttt{f 3 4}), and compiles it as a multi-parameter function.