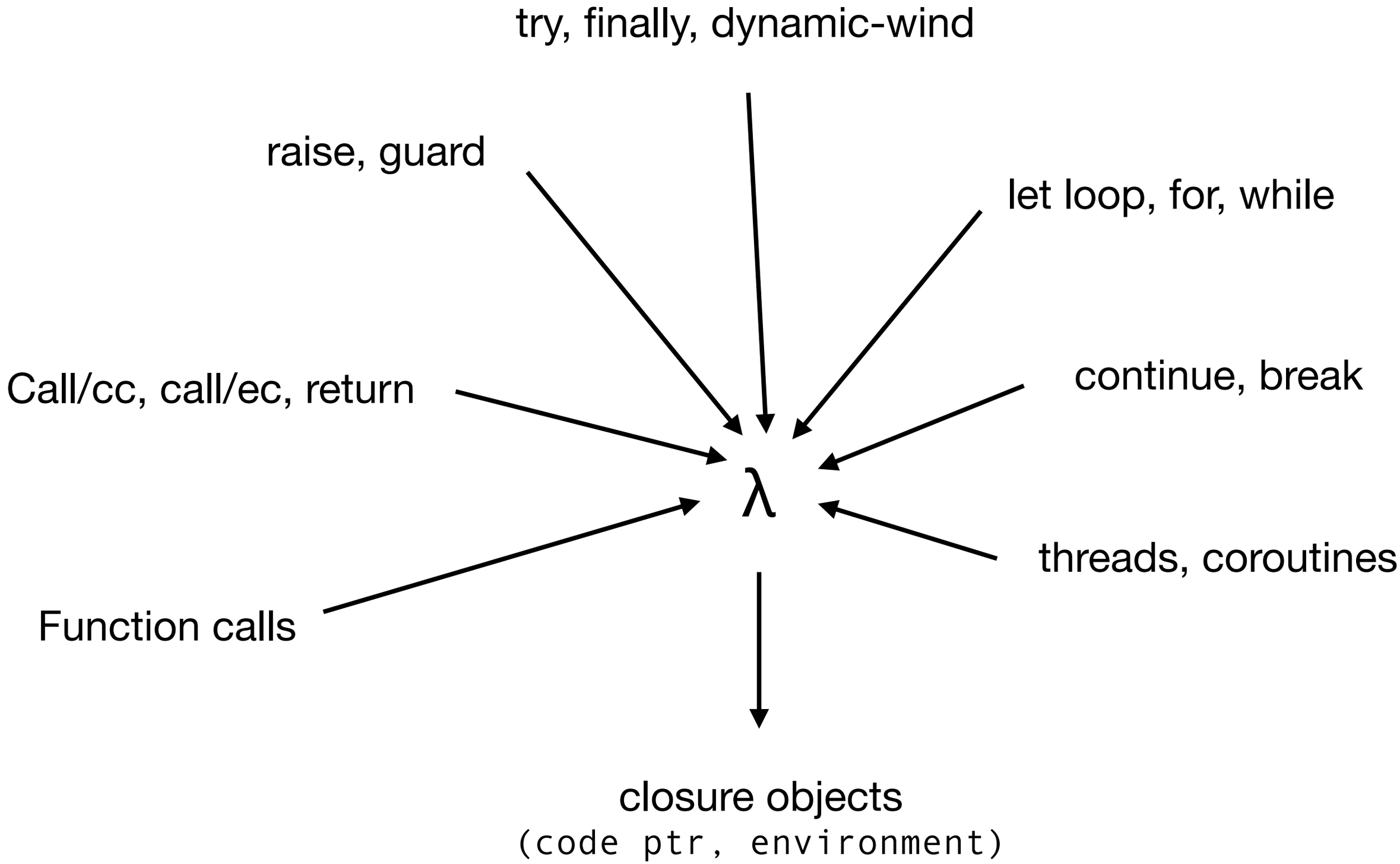
Closure conversion

Compiling λ


try, finally, dynamic-wind
raise, guard
let loop, for, while
Call/cc, call/ec, return
continue, break
λ
Function calls
threads, coroutines
closure objects
(code ptr, environment)

Closure conversion

- In strict CPS, lambdas no longer return.
- Function calls are just first-class `goto`'s that take arguments and carry values for free variables in an environment.
- Closure conversion:
  - Hoists all functions to the top-level.
  - Allocates closures that save the current environment.
  - Function calls explicitly pass the closure's env.
  - Replaces references to free variables with env access.

Closure conversion

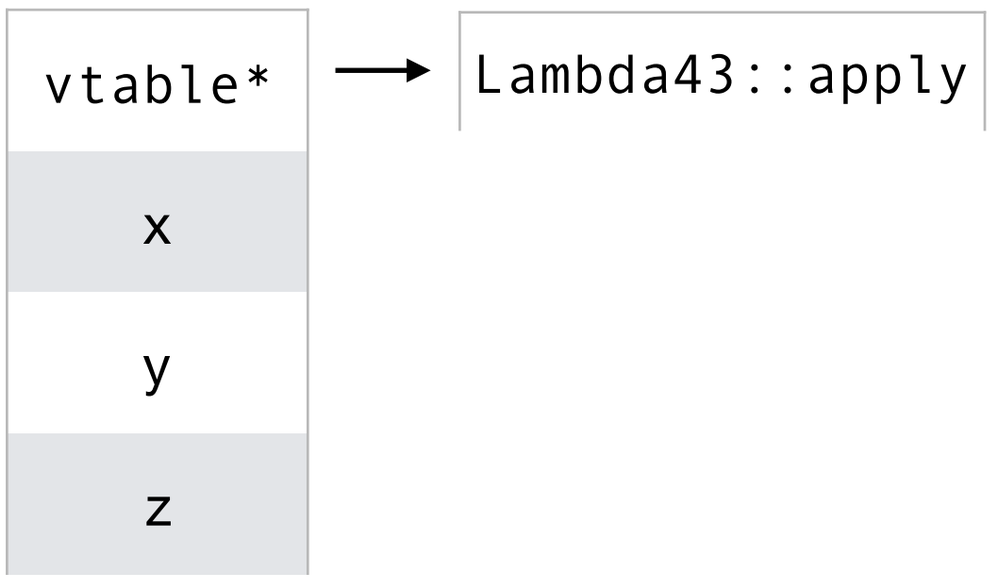
(λ (a b) …) ⟶ new Lambda43(x,y,z);

$\mathcal{FV}$((λ (a b) …)) = {x,y,z}

```
class Lambda43 : public Lam
{
private:
  const u64 x,y,z;
public:
  Lambda43(u64 x, u64 y, u64 z)
    : x(x), y(y), z(z)
  {}

  u64 apply(u64 a, u64 b)
  {
    …
  }
};
```

Closure conversion




```
class Lambda43 : public Lam
{
private:
  const u64 x,y,z;
public:
  Lambda43(u64 x, u64 y, u64 z)
    : x(x), y(y), z(z)
  {}

  u64 apply(u64 a, u64 b)
  {
    …
  }
};
```

Closure conversion:

two principal strategies

- Top-down closure conversion:
    - Traverses the AST and turns lambdas into closures on the way down (computing *environments* as it goes).
    - Produces ***linked environments/closures***.
        - Pros: compact, shared environments; Cons: slower.
- Bottom-up closure conversion:
    - Traverses the AST and turns lambdas into closures on the way back up (computing ***free vars*** as it goes).
    - Produces ***flat environments/closures***.
    - Pros: fast, computes free vars; Cons: more copying.

```
…
   (λ (a)
        …

                    (λ (b c)
                         …

                              (λ (d)
                                   …

                                        (f a c)

…) …) …) …
```

Top-down closure conversion

- As the AST is traversed, a set of locally bound variables are computed.
- A map from non-locally bound variables to their access expressions is maintained so that variables can be looked up in an environment.
- At each lambda: the algorithm lifts the lambda to a first-order C-like procedure, allocates a new closure and its environment, then converts it's body using an updated map for non-locally bound variables (with paths into this newly defined environment).
- Previously allocated environment is linked-to/shared.

…

(λ (a)

…

bound vars: x

(λ (b c)

…

(λ (d)

…

(f a c)

…) …) …) …

```
(proc (main)
 …
  (vector
     lam0
     (prim vector
       x))
 …)
```

env mapping:

```
x -> (vector-ref env0 ‘0)
```

```
(proc (lam0 env0 a)

  …
   (λ (b c)
     …
        (λ (d)
           …
              (f a c)

     …) …) …)
```

```
(proc (main)
 …
  (prim vector
     lam0
     (prim vector
       x))
 …)
```

bound vars: `env0,a,y`

env mapping:

```
x    -> (vector-ref env0 ‘0)
env0 -> (vector-ref env1 ‘0)
a    -> (vector-ref env1 ‘1)
y    -> (vector-ref env1 ‘2)
```

```
(proc (lam0 env0 a)

   …
    (prim vector
          lam1
          (prim vector
                env0
                a
                y))

      …)

(proc (lam1 env1 b c)

    …
      (λ (d)
        …
          (f a c)

    …) …)
```

Bottom-up closure conversion

- As the AST is traversed, free variables are computed.
- At each lambda: 1) the algorithm converts any lambdas under the lambda's body first (and also computes a set of free variables); then 2) it emits code to allocate the lambda's closure/environment and replaces free vars with env access.
- Converting the body of a lambda yields a set of free variables that can be *canonically ordered*.
- Closures are *flat* heap-allocated vectors containing a function pointer and then each free var *in order*.
- Accesses of free variables are turned into a `vector-ref` with the predetermined index.

…

(λ (a)

…

(λ (b c)

…

(λ (d)

…

free vars: a , c , f

(f a c)

…) …) …) …

```
…
  (λ (a)
     …
```

```
(λ (b c)
   …
```

```
(proc (lam14 env d)
  …
  (clo-app
   (prim vector-ref
         env ‘3)    ;f
   (prim vector-ref
         env ‘1)    ;a
   (prim vector-ref
         env ‘2))   ;c
    …)
```

adds first-order proc

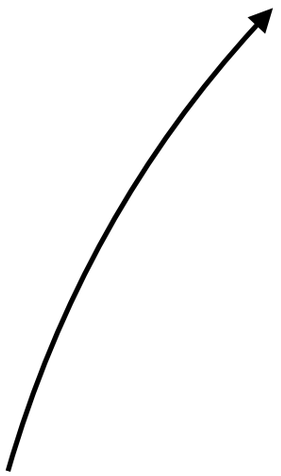
```
(prim vector
      lam14
      a
      c
      f)
```

allocates flat closure

```
…) …) …) …
```

```
…
   (λ (a)
        …
```

```
(λ (b c)
     …
```



```
(prim vector
      lam14
      a
      c
      f)
```

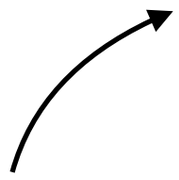
free vars: a f x y

references at closure allocation can remain free

```
…)  …)  …)  …
```

```
(clo-app
   (prim vector-ref
         env ‘3)    ;f
   (prim vector-ref
         env ‘1)    ;a
   (prim vector-ref
         env ‘2))   ;c
```

```
(let ([f-clo (prim vector-ref env ‘3)])
  (let ([f-ptr (prim vector-ref f-clo ‘0)])
    (let ([a (prim vector-ref env ‘1)])
      (let ([c (prim vector-ref env ‘2)])
        (C-style-call f-ptr f-clo a c)))))
```



application: 1) function pointer is accessed from closure

2) closure (`f-clo`) is passed to invoked function ptr

Let’s live code bottom-up closure conversion.