

## Closure conversion

- Objective: all functions should be closed
  - Add all free variables as additional arguments

```
(* Original *)
int fact(int i)
{
  int k = 1;
  while(i) {
    k *= i;
    i--;
  }
  return k;
}
```



---

---

---

---

---

---

---

---

## CPS conversion

```
(* CPS conversion *)
global int f(cont, exnh, i)
{
  int k = 1;
  int break() { cont(k); }
  int loop() {
    if i = 0 then
      break();
    else
      k <- k * i;
      i <- i - 1;
      loop();
  }
  loop();
}
```



---

---

---

---

---

---

---

---

## Introduce free variable arguments

```
(* Closure conversion *)
global int f(cont, exnh, i)
{
  int k = 1;
  int break(k, cont) { cont(k); }
  int loop(i, k, cont) {
    if i = 0 then
      break(k, cont);
    else
      let k = k * i in
      let i = i - 1 in
      loop(i, k, cont);
  }
  in
  loop(i, k, cont);
}
```



---

---

---

---

---

---

---

---

## Dataflow graph

- One vertex for each function, containing
  - Uses: variables used in the function
  - Defs: variables defined by the function
  - Calls: function calls, with defs at the call



---

---

---

---

---

---

---

---

## Building the dataflow graph

- Scan the program
- Functions
  - When a new function definition is found
    - Add a new node to dataflow graph
    - Add all params as defs
- Frames
  - Whenever a global function is found, generate a new frame (this will be used for all escaping vars)
  - For other functions, use the same frame as the parent
- Dataflow
  - For each def
    - Add to "defs"
    - Add to global def table
  - For each use of a variable  $v$ :
    - If  $v$  is a function, add to "calls"
    - Otherwise, add to "uses"



---

---

---

---

---

---

---

---

## Dataflow equation

$$f.\text{live\_in} = \left( f.\text{uses} \cup \bigcup_{g \in f.\text{calls}} g.\text{live\_in} \right) - f.\text{defs}$$

- Solve by iteration

Repeat for all  $f$  forever:

$$f.\text{uses} \leftarrow \left( f.\text{uses} \cup \bigcup_{g \in f.\text{calls}} g.\text{uses} \right) - f.\text{defs}$$



---

---

---

---

---

---

---

---

## Frames

- Variables used by continuations always escape
  - Optimization: store escaping variables in a record, called the frame
  - Use the same frame for every continuation



---

---

---

---

---

---

---

---

## Side-effects

- We *have* to allocate frames to handle side-effects

```
void (*g1)();          void g1(cont);
int (*g2)();           void g2(cont);

void f() {             void f(cont) {
  int i = 0;           int i = 0;
  void h1() { i++; }   void h1(i,cont) { i++; cont() }
  int h2() { return i; } int h2(i,cont) { cont(i); }
  g1 = h1;             g1 = h1(i); // partial application
  g2 = h2;             g2 = h2(i); // partial application
}                      }

g1();                 g1();
g2() --> should return 1 g2() --> should return 1
```



---

---

---

---

---

---

---

---

## Frames

- Use a record to store escaping variables

```
void g1(cont);
void g2(cont);

void f(cont) {
  frame : { i : int };
  frame.i <- 0;
  void h1(frame,cont) { frame.i++; cont() }
  void h2(frame,cont) { cont(frame.i); }
  g1 = h1(frame); // partial application
  g2 = h2(frame); // partial application
}

g1();
g2() --> returns 1
```



---

---

---

---

---

---

---

---

## Frame allocation

- When constructing the dataflow graph, save a frame <fdef,fuse> pair
  - *Fdef is the frame where a variable is defined*
  - *Fuse is the function where the variable is used*
- Each global starts a new <fdef,fuse>
- Each continuation starts a new fuse, but keeps the same fdef
  - *Continuations use the same frame as their parent*
- Local functions do not change <fdef,fuse>



---

---

---

---

---

---

---

---

## Frame linkage

- Goal is that every escaping function has at most one free variable
- Link the frames in a list
  - *Every frame will contain a pointer to the parent frame*



---

---

---

---

---

---

---

---

## Frame dataflow graph

```
let f(cont, i) =  
  let j = 1 in  
  let break(k) = cont(j + k) in  
  let i = 2 in  
  let g(l) = break(i + 1) in  
  g(i)
```

Var	frame
cont	ff
j	ff
k	bf
i	ff
l	gf

f:  
frame = [ff]  
uses = {}  
defs = {j,i}  
calls = {g}

g:  
frame = [gf;ff]  
uses = {i,l}  
defs = {l}  
calls = {break}

break:  
frame=[bf;ff]  
uses = {j, k, cont}  
defs = {k}  
calls = {}



---

---

---

---

---

---

---

---

## Frame assignment

- Build the dataflow
  - Save the  $\langle fdef \rangle$  for each variable definition
  - Save the  $\langle fuse \rangle$  for each function
  - Note that definitions in continuations use the  $\langle fdef \rangle$  of the parent (global) function
- Classify vars as "Simple" or "Frame"
  - If a variable is live in a function with  $\langle fuse \rangle$ , and the variable is defined in frame  $\langle fdef \rangle$ 
    - Initially every var is Simple
    - If  $fuse \neq fdef$  then reclassify as  $Frame(fdef)$



---

---

---

---

---

---

---

---

## Closure construction

- This is what we have:
  - Every variable is classified as Simple, or  $Frame(fdef)$
  - A dataflow graph defining live-in sets for each function
  - A collection of frame definitions  $\langle fdef \rangle$  containing all the variables marked as  $Frame(fdef)$
- We have to:
  - Perform frame allocation for each global function
  - Add live-in variables as extra arguments to each function
  - Generate a closure for each escaping function
  - Add extra arguments to TailCalls



---

---

---

---

---

---

---

---

## Global functions

- For a global function  $f$  with frame  $\langle fdef \rangle$ 
  - Add a function header to allocate a frame
  - Fields get default values
  - For each function parameter, if the param escapes, store it in the frame
  - If the function is nested, add the parent frame as a new parameter
  - Link the parent frame
  - Project all the frames



---

---

---

---

---

---

---

---

## Global functions

```
void f(cont) {
  int i = 1;
  void g(cont) {
    int j = 2;
    void h(cont) {
      int k = 3;
      void z(cont) {
        int l = 4;
        cont(i + j + k + l);
      }
    }
  }
  cont(g);
}

void f(cont) {
  f_frame : { i : int };
  f_frame.i <= 1;
  void g(f_frame, cont) {
    g_frame : { f_frame : f_frame; j : int };
    g_frame.f_frame <- f_frame;
    g_frame.j <= 2;
    void h(g_frame, cont) {
      h_frame : { g_frame : g_frame; k : int };
      h_frame.g_frame <- g_frame;
      let f_frame = g_frame.f_frame in
      h_frame.k <= 3;
      void z(h_frame, cont) {
        z_frame : { h_frame : h_frame };
        z_frame.h_frame <- h_frame;
        let g_frame = h_frame.g_frame in
        let f_frame = g_frame.f_frame in
        let l = 4 in
        cont(f_frame.i + g_frame.j + h_frame.k + l);
      }
    }
  }
}
```



---

---

---

---

---

---

---

---

---

---

## Continuations

- Continuations take their parent's frame as an extra argument
  - *Project all linked frames*
- Local functions
  - *Add all live-in variables as extra parameters, including frames*



---

---

---

---

---

---

---

---

---

---

## Escaping functions

- If a variable use is for a function, generate a closure
- All functions that escape have at most one free var
  - *Global functions have their parent's frame*
  - *Continuations have their parent's frame*
- LetClosure (v, ty, f, a, e)
  - *Allocates a closure for function f with first argument a*



---

---

---

---

---

---

---

---

---

---

## TailCall (f, args)

- Closure-convert the arguments
- If f is a "real" function (not a higher-order function)
  - Add extra arguments to the tailcall



---

---

---

---

---

---

---

---

---

---

## Frame management

- For any definition of variable v
  - If v is Simple, keep the let-definition
  - If v is Frame(fdef), build the let-definition, and add an assignment fdef.v <- v;
- For any use of a variable v
  - If v is a function, build the closure
  - If v is Simple, leave it unchanged
  - If v is Frame(fdef), then project it with LetProject
    - Let v : ty = fdef.v in ...
- SetVar: v : ty <- a; e
  - If v is Frame(fdef), then use fdef.v <- a; e
  - If v is Simple, use a let: let v = a in e



---

---

---

---

---

---

---

---

---

---