Control (cont)

Deian Stefan
(adopted from my & Edward Yang’s CSE242 slides)
Continuations are implicit in your code

• Code you write implicitly manages the future (continuation) of its computation

• Consider: \((2x + \frac{1}{y}) \times 2\)

A. Multiply 2 and x

B. Divide 1 by y

C. Add A and B

D. Multiply C and 2

current computation

rest of the program, current continuation
Continuations are implicit in your code

- Code you write implicitly manages the future (continuation) of its computation

- Consider: \((2 \times x + \frac{1}{y}) \times 2\)
  
  A. Multiply 2 and \(x\)
  
  B. Divide \(1\) by \(y\)
  
  C. Add A and B
  
  D. Multiply C and 2

```
let before = 2*x;
let cont = curResult =>
  (before + curResult) * 2;
cont(1/y)
```
Node.js example

• Implicit continuation:

```javascript
const data = fs.readFileSync('myfile.txt')
console.log(data);
processData(data);
```

• Explicit continuation

```javascript
fs.readFile('myfile.txt', callback)  
function callback (err, data) {
  console.log(data);
  processData(data);
}
```
Continuation passing style (CPS)

• Some languages let you get your hands on the current continuation
  ➤ call/cc (call with current continuation) is used to call a function and give it the current continuation
  ➤ Why is this powerful? A: let’s some inner function bail out and continue program by calling continuation

• Most languages don’t let you get your hands on the current continuation: transform your code to CPS!
Continuation passing style

• Why do we want to do this?
  ➤ Makes control flow explicit: no return!
  ➤ Makes evaluation order explicit

• So? Why should you care about this?
  ➤ IR of a number of languages
  ➤ Turns function returns, exceptions, etc.: single jmp instruction! Can get rid of runtime stack!
To CPS, by example

function zero() {
    return 0;
}

function zero(cc) {
    cc(0);
}

cc is a function (the current continuation)

function zero(cc) {
    cc(0);
}

continue execution by calling cc
To CPS, by example

function fact(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}

function fact(n, cc) {
    if (n == 0) {
        cc(1);
    } else {
        fact(n-1, r => cc(n*r));
    }
}
To CPS, by example

function fact(n, cc) {
    if (n == 0) {
        cc(1);
    } else {
        fact(n-1, r => cc(n*r));
    }
}

fact(3, id) ->
fact(2, r_A => id(3*r_A)) ->
fact(1, r_b => (r_A => id(3*r_A))(2*r_b)) ->
fact(0, r_c => (r_b => (r_A => id(3*r_A))(2*r_b))(1*r_c))
}
To CPS, by example

function twice(f, x) {
    return f(f(x));
}

function cmp(f, g, x) {
    return f(g(x));
}

function twice(f, x, cc) {
    f(x, r => f(r, cc));
}

function twice(f, g, x, cc) {
    g(x, r => f(r, cc));
}
To CPS, by example

```javascript
function twice(f, x) {
    let r = f(x);
    return f(r);
}
```

```javascript
function twice(f, x, cc) {
    f(x, r => f(r, cc));
}
```
To CPS, the rules

• Function decls take extra argument: the continuation

\[
\begin{align*}
\text{function } (x) \{ & \quad \Rightarrow \quad \text{function } (x, \text{cc}) \{ \\
\text{stmt}_1 & \Rightarrow \quad \text{stmt}_1 \\
\text{stmt}_2 & \Rightarrow \quad \text{stmt}_2
\end{align*}
\]

• There are no more returns! Call continuation instead

\[
\begin{align*}
\text{return } x; & \quad \Rightarrow \quad \text{cc}(x); \\
\end{align*}
\]

• Lift nested function calls out of subexpressions

\[
\begin{align*}
\text{let } r = g(x); & \quad \Rightarrow \quad g(x, r \Rightarrow \{ \\
\text{stmt}_1 & \Rightarrow \quad \text{stmt}_1 \; ; \; \text{stmt}_2 \\
\text{stmt}_2 & \Rightarrow \quad \}
\end{align*}
\]
Why is this useful?

• Makes control flow explicit
  ➤ Compilers like this form since they can optimize code
  ➤ One technique: tail-call optimization

• Multithreaded programming

• Event based programming such as GUIs
Continuations are extremely powerful

• Generalization of goto!

• Can implement control flow constructs using continuations

• How do we do if statements?

• How do we do exceptions?
Exceptions w/ continuations

function f() { throw "w00t"; }

try {
    f();
    console.log("no way!");
} catch (e) {
    console.log(e);
}
console.log("cse130 is lit");
Exceptions w/ continuations

1. function f() { throw "w00t"; }
2.
3. try {
4.   f();
5.   console.log("no way!");
6. } catch (e) {
7.   console.log(e);
8. }
9. console.log("cse130 is lit");
Exceptions w/ continuations

success cont = line 5; previous cc = lines 5;9

fail cont = lines 6-8; previous cc = lines 6-9

1. function f() { throw "w00t"; }
2.
3. try {
4.   f();
5.   console.log("no way!");
6. } catch (e) {
7.   console.log(e);
8. }
9. console.log("cse130 is lit");
Control

- Structured programming
- Procedural abstraction
- Exceptions
- Continuations
Fin: the great ideas

**Expressive power (say more with less)**
- First-class functions
- Type inference
- Monads
- Pattern matching
- Exception handling
- Continuations

**Reliability and reuse**
- Type polymorphism
- Modules
- Type classes
- Objects & inheritance