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: (2*x + 1/y) * 2

A. Multiply 2 and x

B. Divide 1 by y

current computation

C. Add A and B

D. Multiply C and 2

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*x + 1/y) * 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:

```
const data = fs.readFileSync(‘myfile.txt’)
console.log(data);
processData(data);
```

- Explicit continuation

```
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;
}
```

→

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, rA => id(3*rA)) ->
fact(1, rb => (rA => id(3*rA))(2*rb)) ->
fact(0, rc => (rb => (rA => id(3*rA))(2*rb))(1*rc)) ->
(rc => (rb => (rA => id(3*rA))(2*rb))(1*rc))(0) ->
(rb => (rA => id(3*rA))(2*rb))(1*0) ->
(rA => id(3*rA))(2*1*0) ->
id(3*2*1*0)
```

To CPS, by example

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

→

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

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

→

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

To CPS, by example

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

→

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

To CPS, the rules

- Function decls take extra argument: the continuation
  - `function (x) {` ➡ `function (x, cc) {`
- There are no more returns! Call continuation instead
  - `return x;` ➡ `cc(x);`
- Lift nested function calls out of subexpressions
  - ➡

```
let r = g(x);
stmt₁
stmt₂
```

```
g(x, r => {
   stmt₁ ; stmt₂
 })
```

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

current cont = line 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”);
```

Exceptions w/ continuations

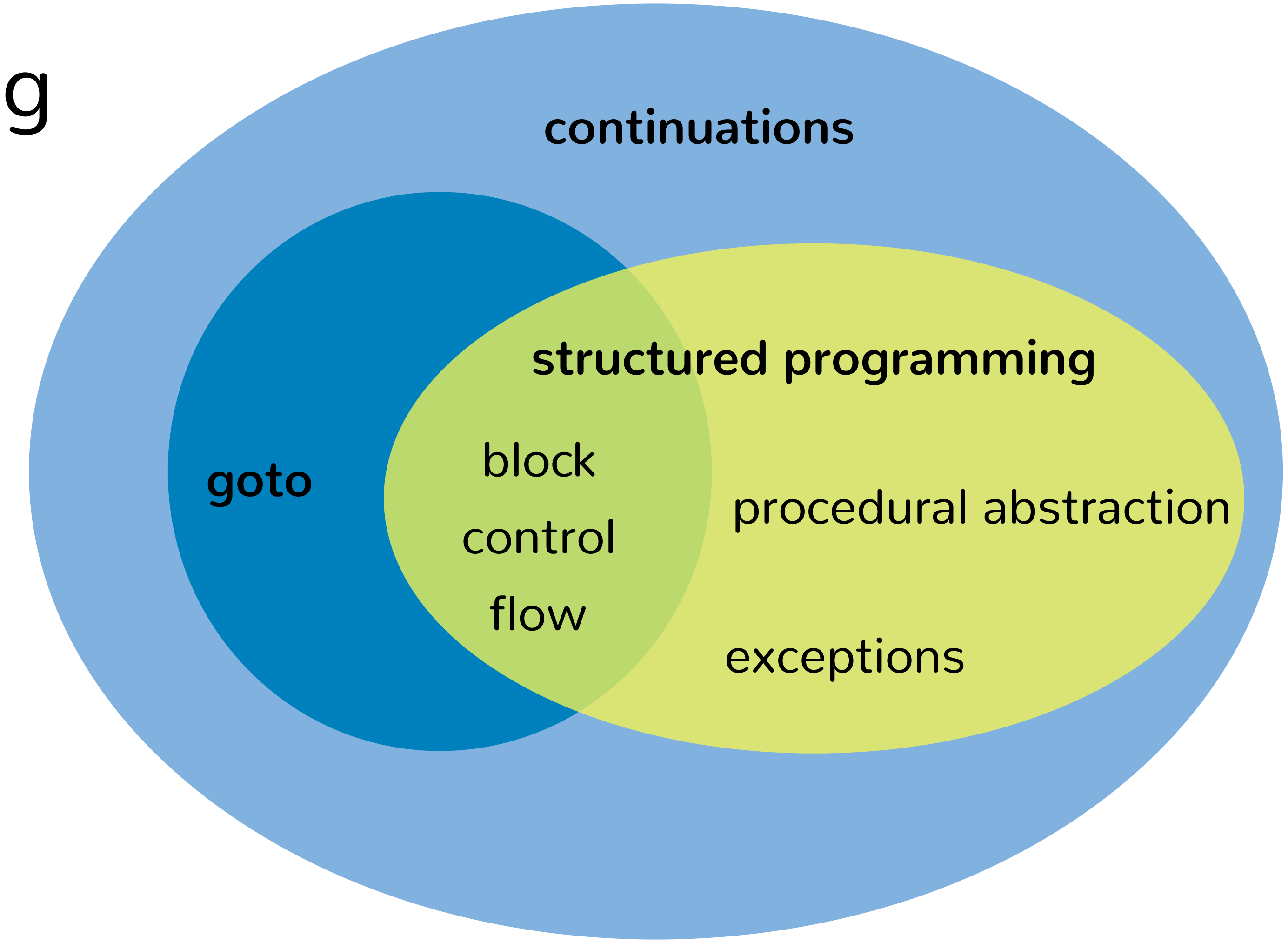
success cont =
line 5;previous cc = lines 5;9

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	Pattern matching
Type inference	Exception handling
Monads	Continuations

Reliability and reuse

Type polymorphism	Type classes
Modules	Objects & inheritance