IA010: Principles of Programming Languages

Control-Flow

Achim Blumensath

blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Continuation passing style

```
let f () {
  let u = input("first: ");
  let v = input("second: ");
  process(u,v)
};
```

Continuation passing style

```
let f () {
  let u = input("first: ");
  let v = input("second: ");
  process(u,v)
};
```

To resume, we need to know:

- where we are
- the result of the expression just computed
- the values of the local variables

Continuation passing style

```
let f () {
  let u = input("first: ");
  let v = input("second: ");
  process(u,v)
};

fun (u) {
  let v = input("second: ");
  process(u,v)
};
```

Continuation passing style

```
let f () {
  let u = input("first: ");
  let v = input("second: ");
  process(u,v)
};

fun (u) {
  let v = input("second: ");
  process(u,v)
};

fun (v) {
  process(u,v)
};
```

Continuation passing style

```
let f () {
  let u = input("first: ");
  let v = input("second: ");
  process(u,v)
};

let f (k) {
  input("first: ",
        fun (u) {
          input("second: ",
                fun (v) {
                  process(u,v,k);
                })
        })
};
```

Example

```
let fac(n) { if n == 0 then 1 else n * fac(n-1) };
```

Example

```
let fac(n) { if n == 0 then 1 else n * fac(n-1) };

let fac_cps(n,k) {
  if n == 0 then
    k(1)
  else
    fac_cps(n-1, fun (x) { k(n*x) })
};
```

Example

```
let fac(n) { if n == 0 then 1 else n * fac(n-1) };

let fac_cps(n,k) {
  equal(n,0,
    fun (c) {
      if c then
        k(1)
      else
        minus(n,1,
          fun (a) {
            fac_cps(a,
              fun (b) { times(n,b,k) })
          })
    })
};
```

Continuations

$$\langle expr \rangle ::= \ldots \mid \texttt{letcc}\ \langle id \rangle\ \texttt{\{}\ \langle expr \rangle\ \texttt{\}}$$

```
letcc k { 1 }
letcc k { k(1) }
letcc k { k(1+2) }
letcc k { 2 + k(1) }
2 + letcc k { k(1) }
3 + letcc k { k(1+2) }
4 + letcc k { 3+k(1+2) }
letcc k { k(1) + k(2) }
letcc k { letcc l { k(1) + l(2) } }
letcc k { letcc l { l(1) + k(2) } }
```

Implementation Issues

- transform the program into continuation-passing style
- more efficient:
  - use a stack
  - make a copy when creating continuations
  - copy it back when calling the continuation

Implementation Issues

- transform the program into continuation-passing style
- more efficient:
  - use a stack
  - make a copy when creating continuations
  - copy it back when calling the continuation

Discussion

- increase in expressive power: user defined control-flow operations
- performance hit
- can lead to spaghetti code

Generators

```
let gen() {
  let n = 0;
  while True {
    yield n;
    n := n+1;
  }
};
```

Generators

```
let gen() {
  let n = 0;
  while True {
    yield n;
    n := n+1;
  }
};
```

```
let gen_return(x) { () };

let gen_helper() {
  let n = 0;
  while True {
    letcc k {
      gen_helper := k;
      gen_return(n)
    };
    n := n+1;
  };
  0
};
let gen() {
  letcc k {
    gen_return := k;
    gen_helper()
  }
};
```

Exceptions

$$\langle expr \rangle ::= \ldots \mid \texttt{try}\ \langle expr \rangle\ \texttt{catch}\ \langle var \rangle\ \texttt{=>}\ \langle expr \rangle$$
$$\mid \texttt{throw}\ \langle expr \rangle$$

```
try 2
catch x => x + 1
=> 2
```

```
try 2 + throw 4
catch x => x + 1
=> 5
```

Exceptions

```
type error = | EmptyList;

let head(lst) {
  case lst
  | []     => throw EmptyList
  | [x|xs] => x
};

try head([])
catch x => 0
```

Exceptions

```
type error   = | NotFound;
type key_val = [ key : a, val : b ];

let lookup(lst : list(key_val), k : a) : b {
  case lst
  | []     => throw NotFound
  | [x|xs] => if x.key == k then
                x.val
              else
                lookup(xs, k)
};
```

Implementation

```
try e catch x => handler   ⟹   letcc k { e(fun (x) { k(handler) }) }
throw e k                  ⟹   k(e)
```

Implementation

```
try e catch x => handler    ⟹    letcc k { e(fun (x) { k(handler) }) }
throw e k                   ⟹    k(e)
```

Efficient Implementation

- remove several stack frames at once and call the handler
- problem 1: how much to remove
- problem 2: calling destructors

Discussion

Exceptions

- (more or less) efficient way to return from deeply nested function calls
- less syntactic overhead
- very hard and error prone to combine with cleanup code and side effects
- creating implicit and non-local control-flow

Error codes

- error prone (easy to forget)
- usually slower (check after each function call)
- more verbose
- control-flow is explicit and local

Algebraic effects

Generalisation of exceptions with ability to **resume** the computation.

```
effect bar : int;

try  3 + bar  catch bar => 1 + abort 5

  ==>  try  3 + throw bar  catch x => 5


try  3 + bar  catch bar => 1 + resume 5

  ==>  3 + 5
```

Algebraic effects

Generalisation of exceptions with ability to **resume** the computation.

```
effect bar : int;

try  3 + bar  catch bar => 1 + abort 5

  ==>  try  3 + throw bar  catch x => 5


try  3 + bar  catch bar => 1 + resume 5

  ==>  3 + 5
```

Everything we said about exceptions also holds for algebraic effects, just more so.

Algebraic effects

Example: implementing `letcc`

```
letcc k { e }
```

Algebraic effects

Example: implementing `letcc`

```
letcc k { e }
```

```
effect get_cc : unit;

let k(x) { () };
try {
  get_cc;
  e
} catch get_cc => {
  k := abort;
  resume;
}
```