Principles of Programming Languages

Lecture 10 Continuations

Albert-Ludwigs-Universität Freiburg

Peter Thiemann
University of Freiburg, Germany
thiemann@informatik.uni-freiburg.de

09 July 2018

Plan

Continuations

- concept for expressing exceptions, backtracking, coroutines, multi-threading, ...
- became popular with server-side web-programming
- "enforced" by reactive programming

Continuation-Passing Style (CPS)

Places syntactic restrictions on programs

- all functions are tail-recursive
- functions never return
- each function has one or more **continuation parameters**, each of which is a function
- to return a value, a function invokes a continuation and passes the return value(s) as a parameter.

Plan

Motivation: Exceptions

Multiply a tree of numbers

```
product xt =
  if null xt
  then 1
  else product (left xt) * value xt * product (right xt)
```

- inefficient if `xt` contains zero

Trying to improve . . .

```
product1 xt =
  if null xt
  then 1
  else if value xt == 0
  then 0
  else product (left xt) * value xt * product (right xt)
```

- better, but still many useless multiplications

Rewrite in CPS

```
product2 xt =
  prod xt (\x -> x)
  where
  prod xt c =
    if null xt
    then c 1
    else if value xt == 0
    then c 0
    else prod (left xt) (\l -> prod (right xt) (\r -> c (l * r * value xt)))
```

- same as `product1`, but in CPS
- c argument is continuation
- now we can exploit the presence of continuations

Changing the flow of control in CPS

```
product3 xt =
  prod xt (\x -> x)
  where
  prod xt c =
    if null xt
    then c 1
    else if value xt == 0
    then 0     -- do NOT invoke the continuation c
    else prod (left xt) (\l -> prod (right xt) (\r -> c (l * r * value xt)))
```

- deliberate violation of CPS
- achieves the desired effect
- useful, but clumsy

Call/cc — a Control Operator

```
product3 xt =
  call/cc (\ abort ->
    let prod xt =
      if null xt
      then 1
      else if value xt == 0
      then abort 0
      else prod (left xt) * value xt * prod (right xt)
    in prod xt
```

- Call/cc = call with current continuation
- Applies its argument to the current continuation
- Advantage: no need to write programs in CPS
- Disadvantage: some experience/insight needed

Plan

A Problem

Consider the subset sum problem which is a specialized version of the knapsack problem:
You are given

- a positive integer C (the weight that you can carry) and
- a list of positive integers (the weights of items you want to carry).

Is there a subset of the items the weights of which add up to C? (This problem is known to be NP-complete.)

An Implementation (pseudo code)

```
subsetsum target items =
  let work path target items =
    if target == 0
    then RESULT path
    else if null items
    then FAIL
    else if (head items) <= target
    then TRY (work (head items : path) (target - head items) (tail items))
         ANDTHEN work path target (tail items)
    else work path target (tail items)
  in
    work [] target items
```

Observations

Primitives in pseudo code

- `RESULT` announces a result.
- `FAIL` declares the current invokation to fail.
- `TRY ...ANDTHEN ...` searches for a result in the first argument and then in the second.

Observations

Primitives in pseudo code

- `RESULT` announces a result.
- `FAIL` declares the current invokation to fail.
- `TRY ...ANDTHEN ...` searches for a result in the first argument and then in the second.

Implementation with continuations

- each function has **two** continuations `succ` and `fail`
  - invoke `succ` to indicate success and return a result
  - invoke `fail` to indicate failure
- implement `TRY ...ANDTHEN ...` by nesting continuations

Double-barreled CPS

```
subsetsum1 target items =
  let work path target items succ fail =
    if target == 0
    then succ path
    else if null items
    then fail ()
    else let hi = head items in
    if hi <= target
    then work (hi : path) (target - hi) (tail items) succ (\ () ->
              work path target (tail items) succ fail)
    else work path target (tail items) succ fail
  in
    work [] target items (\ x -> True) (\ () -> False)
```

Comments

- Using the success continuation in this example is overblown.
- Replacing `succ path` with `True` yields the same behavior.
- However, the success continuation may be used to compose a list of all results or to compute a best approximation:

List of all results

```
subsetsum1 target items =
  let work path target items succ fail =
    if target == 0
    then succ path fail
    else if null items
    then fail ()
    else let hi = head items in
    if hi <= target
    then work (hi : path) (target - hi) (tail items) succ (\ () ->
         work path target (tail items) succ fail)
    else work path target (tail items) succ fail
  let results = ref []
  in work [] target items (\ path fail -> results := path : !results;
                                           fail ())
                          (\ () -> return !results)
```

Best approximation

```
subsetsum2 target items =
  let work path target items succ fail =
    if target == 0 || null items
    then succ target path fail
    else let hi = head items in
    if hi <= target
    then work (hi : path) (target - hi) (tail items) succ (\ () ->
         work path target (tail items) succ fail)
    else work path target (tail items) succ fail;
  let best = ref target;
  let result = ref [];
  in work [] target items (\ rest path fail ->
                            if rest < !best then (best := rest; result := path);
                            fail ())
                          (\ () -> return !result)
```

Plan

Motivation: Coroutines

Coroutines

- program components like subroutines
- on equal footing, without caller-callee hierarchy
- exactly one coroutine is active at any instance
- active coroutine can yield control (with parameters) to another, which resumes from where it yielded previously

Motivation: Coroutines

UNI FREIBURG

Coroutines

- program components like subroutines
- on equal footing, without caller-callee hierarchy
- exactly one coroutine is active at any instance
- active coroutine can yield control (with parameters) to another, which resumes from where it yielded previously

What are they good for

- Coroutines are well suited for implementing programming patterns such as cooperative tasks, iterators, infinite lists, and pipes.
- Available in Python, Lua, C#, etc

Coroutines, example use

```
/** run-length decompression */
void decompress () {
    while ((c = getchar()) != EOF) {
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                emit(c);
        } else
            emit(c);
    }
    emit(EOF);
}
```

Scanner

```
void scanner () {
    while ((c = getchar()) != EOF) {
        if (isalpha(c)) {
            do {
                add_to_token(c);
                c = getchar();
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
    }
}
```

Problem: combine decompress and scanner

- simple code in separation
- task: scanner for compressed documents
- standard approach: rewrite one of the functions
- not required with coroutines
- (here: symmetric coroutines; simpler with asymmetric coroutines as in Python)

Combining with coroutines

```
void scanner (COROUTINE producer) {
    while ((c = yield(producer)) != EOF) {
        if (isalpha(c)) {
            do {
                add_to_token(c);
                c = yield(producer);
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
    }
}
```

Combining with coroutines, part 2

```
void decompress (COROUTINE consumer) {
    while ((c = getchar ()) != EOF) {
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                yield(consumer, c);
        } else
            yield(consumer, c);
    }
    yield(consumer, EOF);
}
```

Driver

```
run (COROUTINE producer, COROUTINE consumer) {
  do {
    c = yield (producer);
    yield (consumer, c);
  } while (c != EOF);
}

...
COROUTINE producer = make_coroutine (decompress);
COROUTINE consumer = make_coroutine (scanner);
COROUTINE driver = make_coroutine (run);

driver (producer, consumer);
...
```

Implementing Coroutines with Call/cc

```
running = ref (ref Nothing)
make_coroutine f =
  let mycont = ref Nothing
  in  \ x ->
      running := Just mycont
      case cont of
        Nothing -> f x
        Just ff -> ff x
yield g y =
  call/cc (\ resume ->
    !running := Just resume;
    g y)
```

Idea: represent each coroutine state by the coroutine's current continuation.

Plan

Motivation: Threads

- native threads vs. simulated threads. The former rely on the operating system and may be executed on different processors. The latter simulate concurrency inside of a sequential process.
- preemptive vs. cooperative. In each thread implementation, a scheduler determines which thread becomes active next. With preemption, the scheduler runs at regular time intervals. It suspends the currently active thread and selects another thread from a pool of suspended threads to run in the next time slice. With cooperative threading, a thread remains active until it explicitly relinquishes control or until it gets blocked due to an I/O operation.

Implementing cooperative threads with call/cc

- Simple user-level implementation of simulated, cooperative threads with call/cc.
- A thread yields to the scheduler.
- Threads communicate exclusively via shared state. They cannot receive parameters or return values while they are running.

A typical thread interface

```
spawn :: (Unit -> Unit) -> Thread
yield :: Unit -> Unit
terminate :: Unit -> Unit
```

A simple thread implementation

```
currentThread = NULL
runQueue = emptyQueue

spawn f =
  enqueue (runQueue, makeThread f)

makeThread f =
  { cont =
      \ () ->
        f (); terminate ()
    ...
  }
```

A simple thread implementation (cont)

```
terminate () =
  scheduleThread (dequeue (runQueue))

scheduleThread (thread) =
  currentThread = thread;
  currentThread.cont ()

yield () =
  call/cc (\ mycont ->
    currentThread.cont = mycont;
    enqueue (runQueue, currentThread);
    scheduleThread (dequeue (runQueue));
    )
```

Plan

1. Continuations
   - Motivation: Exceptions
   - Motivation: Backtracking
   - Motivation: Coroutines
   - Motivation: Threads
   - Implementing first-class continuations

Implementing first-class continuations
via interpretation

Syntax

$$e \quad ::= \quad x \mid \lambda x.e \mid e\ e \mid 0 \mid e{+}e \mid \mathit{if}\ e\ e\ e \mid \texttt{call/cc}$$

Implementing first-class continuations
via interpretation

UNI FREIBURG

Syntax

$$e \quad ::= \quad x \mid \lambda x.e \mid e\ e \mid 0 \mid e{+}e \mid \mathit{if}\ e\ e\ e \mid \texttt{call/cc}$$

Semantic domains

$$\begin{array}{rlcl}
y \in & \mathsf{Val} & = & \mathbf{Z} + (\mathsf{Val} \to \mathsf{Comp}) \\
\kappa \in & \mathsf{Cont} & = & \mathsf{Val} \to \mathsf{Answer} \\
& \mathsf{Comp} & = & \mathsf{Cont} \to \mathsf{Answer} \\
\rho \in & \mathsf{Env} & = & \mathsf{Var} \to \mathsf{Val} \\
& \mathcal{E} & : & \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Comp}
\end{array}$$

Definition of the interpreter

$$
\begin{array}{lcl}
\mathcal{E}[\![x]\!]\rho\kappa & = & \kappa(\rho(x)) \\
\mathcal{E}[\![\lambda x.e]\!]\rho\kappa & = & \kappa(\lambda y.\lambda\kappa.\mathcal{E}[\![e]\!]\rho[x \mapsto y]\kappa) \\
\mathcal{E}[\![e_1\ e_2]\!]\rho\kappa & = & \mathcal{E}[\![e_1]\!]\rho(\lambda y_1.\mathcal{E}[\![e_2]\!]\rho(\lambda y_2.y_1\ y_2\ \kappa)) \\
\mathcal{E}[\![0]\!]\rho\kappa & = & \kappa(0) \\
\mathcal{E}[\![e_1{+}e_2]\!]\rho\kappa & = & \mathcal{E}[\![e_1]\!]\rho(\lambda y_1.\mathcal{E}[\![e_2]\!]\rho(\lambda y_2.\kappa(y_1{+}y_2))) \\
\mathcal{E}[\![\mathit{if}\ e_1\ e_2\ e_2]\!]\rho\kappa & = & \mathcal{E}[\![e_1]\!]\rho(\lambda y.\texttt{if}\ y\ (\mathcal{E}[\![e_2]\!]\rho\kappa)\ (\mathcal{E}[\![e_3]\!]\rho\kappa)) \\
\mathcal{E}[\![\texttt{call}/\texttt{cc}]\!]\rho\kappa & = & \kappa(\lambda f.\lambda\kappa.f(\lambda y.\lambda\kappa'.\kappa y)\kappa) \\
\mathcal{E}[\![\texttt{call}/\texttt{cc}\ e]\!]\rho\kappa & = & \mathcal{E}[\![e]\!]\rho(\lambda f.f(\lambda y.\lambda\kappa'.\kappa y)\kappa)
\end{array}
$$

- Interpreter $\mathcal{E}$ is written in CPS
- BTW, internalizes call-by-value
- Alternatives
  - transform the program to CPS and run it directly
  - implement call/cc natively