Yield, the Control Operator
Exploring Session Types

Roshan P. James and Amr Sabry

Indiana University, Bloomington

CW — September 24, 2011

Continuations: the idea

The sixties and seventies . . .

- Ability to access the rest of the computation as a first-class object.

```
>>> 2 + callcc (\k -> 3 + k 4)
  = 2 + (3 + (abort (2 + 4)))
  = 6
```

- When we have a small complete program, the rest of the computation is well-defined and it is sensible to give one part of the program control over the entire computation.

- In general, there is not even a notion of a "complete program" and it is not acceptable to give some part of the computation control over other arbitrary computations with which it is interacting.

Delimited continuations: the idea

The eighties and nineties ...

- Add a delimiter that marks the "beginning" of the rest of the computation
- All control actions within the dynamic scope of the delimiter are interpreted "up to the occurrence of the delimiter"

```
f a b = shift (\k -> a + k b))

t     = 1 + reset (2 + f 3 4)
      = 1 + reset (2 + shift (\k -> 3 + k 4))
      = 1 + reset (3 + (2 + 4))
      = 10
```

Typed delimiters: fixed answer type

- Let's start with the (reasonable) idea that reset takes a (sub)computation of a given type and returns a value of that type.
- Depending on the details of the surrounding language, reset would have a type like:
  $(() \to a) \to a$
  or
  $CC\ a \to a$
  or
  $CC\ a \to CC\ a$
  or
  $CC\ a\ a \to CC\ w\ a$

Quiz: what's the type of reset in this example?

```
t3 = reset (1 + ...)
```

- What is the type of the subcomputation $(1 + ...)$?

Quiz: what's the type of reset in this example?

```
t3 = reset (1 + ...)
```

- What is the type of the subcomputation (1 + ...) ?
- bool

Quiz: what's the type of reset in this example?

```
t3 = reset (1 + ...)
```

- What is the type of the subcomputation (1 + ...) ?
- bool
- seriously bool

Quiz: what's the type of reset in this example?

```
t3 = reset (1 + ...)
```

- What is the type of the subcomputation (1 + ...) ?
- bool
- seriously bool

```
t3 = reset (1 + shift (\k -> 2 == (k 3)))
```

- As soon as we start executing, the continuation $1 + \Box$ is captured and replaced by the continuation $2 == \Box$
- So the reset subcomputation actually returns a bool and the expression evaluates to *false*

Huh?

- So, you're telling me that if I write:

```
t3 = reset (1 + f 2 3)
```

- then depending on what f is, this might return an int or a bool ?

Huh?

- So, you're telling me that if I write:

```
t3 = reset (1 + f 2 3)
```

- then depending on what f is, this might return an int or a bool ?
- Is this a bug or a feature?

Let's consider it a feature for now

Consider *printf* :

- Depending on the formatting directives, we want to return a *string* or a function $int \rightarrow string$ or a function $string \rightarrow int \rightarrow string$ etc.
- Hard to write a type-safe version but possible using shift and reset:
  - Code is essentially: *reset formatter*
  - If the current answer type is $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow a$ and we encounter a formatting directive for a t, change the answer type to $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t \rightarrow a$

A delimiter with many types

Danvy and Filinski (1989) proposed a type system with judgments $\Gamma, d_1 \vdash e : a, d_2$ that can be interpreted as follows:

- Assuming the delimiter has type d_1 the evaluation of e either:
- returns to its immediate context with a value of type a, or
- performs a control action which captures the d_1-delimited continuation and replaces it by a d_2-delimited continuation
- A modern presentation would be using indexed (or parameterized) monads in which computations have type $CC\ d_1\ d_2\ a$
- Polymorphically (Asai and Kameyama)

Kiselyov's implementation

```
reset  :: CC sigma tau sigma -> CC a a tau
shift  :: ((tau -> CC t t a) -> CC s b s) -> CC a b tau
run    :: CC tau tau tau -> tau

-- t3 = reset (1 + shift (\k -> 2 == (k 3)))
t3 = run $ reset (bind
                  (shift (\k -> bind
                                (k 3)
                                (\w -> ret (2 == w))))
                  (\v -> ret (1+v))
```

The outer *bind*-expression (the reset subcomputation) has type *CC Int Bool Int*

Pause

- Let's keep that background in mind and let's explore other ways to model delimited continuations

- Rememeber that even though we can typecheck these strange type-shifting examples, someone who views types as specifications should be uncomfortable

Pause

- Let's keep that background in mind and let's explore other ways to model delimited continuations
- Rememeber that even though we can typecheck these strange type-shifting examples, someone who views types as specifications should be uncomfortable
- I am

Yield

- Many variants in Ruby, Python, C#, JavaScript, etc.
- We previously proposed a generalized version with two operators *yield* and *runY*
- *runY* is the delimiter
- *yield* captures the continuation up to the delimiter and returns a suspension containing a yielded value and a resumer
- if the resumer is invoked, the computation resumes from the point of the capture of the continuation

Change of perspective

- Equivalent to shift and reset but with a different "feel"

- programming with *yield* and *runY* feels like programming with processes that suspend/resume and that communicate with their "context" by sending and receiving messages

Example: tree walking

```
data Tree = Leaf | Node Int Tree Tree

-- process that generates node labels
inOrder tr = runY (traverse tr)
  where traverse Leaf = return ()
        traverse (Node label left right) =
          do traverse left; yield label; traverse right

-- process that performs an action for each label
useInOrder tr = foreach (inOrder tr) print
  where
    foreach (Result r) _ = return r
    foreach (Susp v resumer) action =
      do i <- action v; foreach (resumer i) action
```

Fixed types: the idea

- Computations have type *CC i o r*
- Running them produces iterators of type *Iterator i o r*

```
data Iterator i o r = Result r
                    | Susp o (i -> Iterator i o r)
```

- In particular *inOrder* :: *Iterator* () *Int* ()

Fixed types: the idea

- Computations have type *CC i o r*
- Running them produces iterators of type *Iterator i o r*
- The type *o* is what the generator sends to the code surrounding the delimiter (an interpreter for *o*-values!)
- The interpreter receives the values of type *o* and processes them producing for each *o*-value an *i*-value
- The *i*-values are sent to the resumer
- The generator uses all the *i*-values to calculate its final answer of type *r*

Example: Dynamic binding, mutable variables, and stack inspection

```
type Dyn t r = CC t (Cmd t) r

data Cmd t   = Lookup Name
             | Assign Name t
             | Inspect Name (t -> Dyn t t)
```

Example: Asynchronous workflows

```
type AsyncProc a      = CC OperationResult Operation a

data Operation        = WebRequest String | WriteDb  ...
data OperationResult  = WebResult String  | DbResult ...
```

Fixed types

The types of yielded values and the types of resumers are fixed throughout the computation

```
data Iterator i o r = Result r
                    | Susp o (i -> Iterator i o r)

instance Monad (CC i o)
yield :: o -> CC i o i
runY  :: CC i o r -> Iterator i o r
```

Change of perspective

- We usually view the delimiter as having the fixed type *Iterator* i o r
- View *Iterator* i o r as a varying type: receive o, respond with i, receive o, respond with i, and so on, until you receive r.
- A type for a session between two processes
- A natural way to generalize the type to: receive o_1, respond with i_1, receive o_2, respond with i_2, and so on, until you receive r.

Session types for Ruby-style *yield*

- Let's start with a restricted version of yield, similar to what's available in Ruby
- Restriction means that the resumer is only exposed to *foreach* or in other words that the only context allowed for a computation is $foreach\ (runY\ \Box)$
- Think one-shot, linearly used, continuations

Session types for Ruby-style *yield*

- No need to send the continuation back and forth
- Generator can suspend itself, sending an output to the consumer (*foreach* context), and keeping its continuation implicit
- The consumer (*foreach* loop) can use the output from the generator and then resume it with an arbitrary value.
- Generator and consumer can run in separate threads and communicate via a shared channel
- Direct connection to session types

Pucella and Tov's embedding of session types in Haskell

```
data Z          -- zero
data S a        -- successor

data (:!:) a r -- output
data (:?:) a r -- input
data (:+:) r s -- choice
data (:&:) r s -- offer
data Rec r      -- recursive definition
data Var v      -- recur to index v
data End        -- end of session

data Cap e r    -- environment e with session r
```

Session types in Haskell

```
newtype Session s s' a

close :: Session (Cap e End)               ()               ()
sel1  :: Session (Cap e (r :+: s))         (Cap e r)        ()
sel2  :: Session (Cap e (r :+: s))         (Cap e s)        ()
enter :: Session (Cap e (Rec r))           (Cap (r,e) r)    ()
zero  :: Session (Cap (r,e) (Var Z))       (Cap (r,e) r)    ()
suc   :: Session (Cap (r,e) (Var (S v)))  (Cap e (Var v))  ()
offer :: Session (Cap e r) u a ->
         Session (Cap e s) u a ->
         Session (Cap e (r :&: s)) u a

inC   :: Session (Cap e (a :?: r)) (Cap e r) a
outC  :: a -> Session (Cap e (a :!: r)) (Cap e r) ()
yield :: o -> Session (Cap e (o :!: i :?: r)) (Cap e r) i
```

Session types in Haskell

```
newtype Rendezvous r

newRendezvous :: IO (Rendezvous r)
accept        :: Rendezvous r -> Session (Cap () r ) () a -> IO a
request       :: Dual r r' =>
                 Rendezvous r -> Session (Cap () r') () a -> IO a
```

Yield example

```
-- process that generates all Fibonacci numbers
fib :: Session (Cap e (Rec (End :&: (Int :!: () :?: Var Z))))
               ()
               ()


fib = enter >>> loop 0 1
  where loop a b = offer
                     close
                     (yield a >>> zero >>> loop b (a+b))
```

Yield example

```
-- process that consumes the first 20 Fibonacci numbers
useFib :: Session (Cap e (Rec (End :+: (Int :?: () :!: Var Z))))
                  ()
                  ()

useFib = enter >>> loop 0
  where loop n | n > 20 = sel1 >>> close
               | otherwise = sel2 >>>
                             inC >>>= \ (i :: Int) ->
                             io (print i) >>>
                             outC () >>>
                             zero >>>
                             loop (n+1)

-- putting producer and consumer together
runFib = do rv <- newRendezvous
            forkIO (accept rv fib)
            request rv useFib
```

Tree walking again

```
fib :: Session (Cap e (Rec (End :&: (Int :!: () :?: Var Z))))
               ()
               ()

inOrder :: Tree ->
           Session
             (Cap () (Rec (End :+: (Int :!: () :?: Var Z))))
             ()
             ()
```

Assessment

- In the case of Ruby-style yield, we have a clear process-like view of delimited continuations
- Session types provide a neat alternative to the fact that the type of the delimiter changes during a computation
- Types feel like specifications again: I am happy
- Not immediately extensible to full delimited continuations

Assessment

- In the case of Ruby-style yield, we have a clear process-like view of delimited continuations
- Session types provide a neat alternative to the fact that the type of the delimiter changes during a computation
- Types feel like specifications again: I am happy
- Not immediately extensible to full delimited continuations except that Oleg says he did it 5 years ago!

Assessment

- In the case of Ruby-style yield, we have a clear process-like view of delimited continuations
- Session types provide a neat alternative to the fact that the type of the delimiter changes during a computation
- Types feel like specifications again: I am happy
- Not immediately extensible to full delimited continuations except that Oleg says he did it 5 years ago!
- Seriously Kiselyov and Shan's "Substructural type system for delimited continuations" is quite relevant but work is needed to formalize the connection to session types
- Probably this idea is scattered in many different places. . .