Delimited continuations in operating systems

Oleg Kiselyov (FNMOC)
oleg@pobox.com

Chung-chieh Shan (Rutgers University)
ccshan@rutgers.edu

1 Continuations are the meanings of evaluation contexts

A context is an expression with a hole.

```
... print( 42 + 2 * 3 ) ...
```

An undelimited evaluation context means an undelimited continuation (a map from integers to outcomes)

A delimited evaluation context means a delimited continuation (a map from integers to integers: adding 42)

2 Continuations abound in operating systems

The context of a system call is a continuation delimited by the process boundary.

Operating system	User process	System call		
`... schedule(`	`main() {...`	`read(...)`	`...}`	`) ...`

The execution context of the user process is captured in the request and awaits a response from the operating system

The operating system may invoke the delimited continuation (after a `read`)
discard the delimited continuation (`exit`)
duplicate the delimited continuation (`fork`) ...see §7

Other examples of delimited continuations:

- event handlers
- Web forms
- coroutines
- process snapshots (checkpoints)
- **less known: traversing a data structure**
  ⟶ navigating and updating a file system with snapshots and transactions

3 Continuations let us convert an enumerator to a cursor

An enumerator for a data structure feeds each element of the data structure to a callback function provided by the client.

Client	Enumerator	Callback		
`...`	`for each x in list {`	`print(x)`	`}`	`...`

The callback function can create a cursor by capturing the current state of the enumerator as a delimited continuation. The delimiter is `reset` and the capturer is `shift` below.

Client	Enumerator	Callback		
`cursor1 = reset(`	`for each x in list {`	`shift k. (x,k)`	`}`	`)`

```
print(curr(cursor1))
cursor2 = next(cursor1)
print(curr(cursor2))
```

Using cursors, we can navigate the data structure and update it *nondestructively*: each update creates a new snapshot, so older versions are intact and concurrent traversals are isolated from each other's updates.

4 Continuations let us derive file-system transactions

A hierarchical file system is a tree of strings (could be richer). In Haskell:

```
type Name = String
type Contents = String
data Term = File Contents | Folder (Map Name Term)
```

Nodes in the file system are connected by edges.

```
data Path = Down | DownTo Name | DownToN Int | Up | Next
```

The enumerator `traverse` exchanges nodes and edges with the callback.

```
traverse :: Monad m => (Path -> Term -> m (Maybe Term, Path))
                    -> Term -> m Term
```

Converting `traverse` to a cursor lets us navigate and update a Unix-like file system!

5 The Zipper File System prototypes these ideas

ZFS is a transactional file system that supports

- **multiple undo and snapshots;**
- **strongest isolation mode: repeatable read;**
- pervasive copy-on-write for files and directories;
- cyclic directory references;
- lazy infinite hierarchies, such as game trees.

ZFS *explicitly* uses delimited continuations as a uniform framework for

- all file system operations;
- multiprocessing;
- exception handling.

The ZFS file server currently runs on a host operating system such as Unix.

- The host's virtual memory contains all data and mediates all IO.
- Demo clients connect via a simple TCP protocol. NFS, 9P, etc. can be added.

6 See these features in our demo

Familiar commands

```
/> cd d1
/d1/> ls
  d11/
  f113
```

Also: `cat`, `mkdir`, `touch`, `echo`, `rm`, `mv`, `cp`. `cp` copies both files and directories. Actual copying waits until changes are made.

Traversal to the next node

```
/d2/> next
/> next
/f11/> next
/> next
/f12/> next
/> next
/> next # done: stuck at the root
```

Transactional semantics

```
/> ls
  d1/    d2/    f11    f12
/> ls d1
  d11/   f113
/> rm d1 # can remove whole trees
/> ls
  d2/    f11    f12
/> refresh # revert to current
           # public version
/> ls
  d1/    d2/    f11    f12
/> ls d1
  d11/   f113
/> rm d1
/> commit # publish our changes
```

Another client won't see any of our changes until we `commit` them.

7 Continuations enable multiprocessing

The Zipper File System serves multiple simultaneous clients in a single host thread.

- It contains a main event loop, an "interrupt" handler, and a system-call interface.
- The execution context of each client process is a delimited continuation.

Process isolation in types

```
run'process :: (∀m. Monad m => Prompt r (OSReq r m) -> CC r m (OSReq r m))
            -> Prompt r (OSReq r IO) -> CC r IO (OSReq r IO)
```

Static types guarantee that a process cannot affect the external world or any global state!

- No need for any thread synchronization, such as locking.
- Transactional semantics for free!

Processes and OS requests

Every client request except `exit` contains the suspended execution context of the client.

```
data OSReq r m
  = OSRDone
  | OSRRead          (CC r m String          -> CC r m (OSReq r m))
  | OSRWrite String  (CC r m ()              -> CC r m (OSReq r m))
  | OSRCommit Term   (CC r m ()              -> CC r m (OSReq r m))
  | OSRefresh        (CC r m (FSZipper r m)  -> CC r m (OSReq r m))
```

To make a request, the client captures its own delimited continuation.

Operating system	User process	System call		
`... run'process(`	`...`	`shift k. (OSRRead k)`	`...`	`) ...`

Delimited continuations do the right thing for free

http://okmij.org/ftp/Computation/Continuations.html#zipper-fs