Delimited continuations in operating systems

Oleg Kiselyov[1] and Chung-chieh Shan[2]

[1] FNMOC oleg@pobox.com
[2] Rutgers University ccshan@rutgers.edu

Abstract. *Delimited continuations* are the meanings of delimited evaluation contexts in programming languages. We show they offer a uniform view of many scenarios that arise in systems programming, such as a request for a system service, an event handler for input/output, a snapshot of a process, a file system being read and updated, and a Web page. Explicitly recognizing these uses of delimited continuations helps us design a system of concurrent, isolated transactions where desirable features such as snapshots, undo, copy-on-write, reconciliation, and interposition fall out by default. It also lets us take advantage of efficient implementation techniques from programming-language research. The Zipper File System prototypes these ideas.

1 Introduction

One notion of context that pervades programming-language research is that of *evaluation contexts*. If one part of a program is currently running (that is, being evaluated), then the rest of the program is expecting the result from that part, typically waiting for it. This rest of the program is the evaluation context of the running part. For example, in the program "$1 + 2 \times 3$", the evaluation context of the multiplication "2×3" is the rest of the program "$1 + $".

The meaning of an evaluation context is a function that maps a result value to an answer. For example, the meaning of the evaluation context "$1 + $" is the increment function, so it maps the result value 6 to the answer 7. Similarly, in a program that opens a file and summarizes its contents, the meaning of the evaluation context of the opening of the file is a function that maps a handle for a file to a summary of its contents. This function is called a *continuation*.

A continuation is *delimited* when it produces an intermediate answer rather than the final outcome of the entire computation. For example, the increment function is a delimited continuation when taken as the meaning of "$1 + $" in the program "print$(1 + 2 \times 3)$". Similarly, we treat a function from file handles to content summaries as a delimited continuation when we view the summarization program as part of an operating system that reaches its final outcome only when the computer shuts down months later. The *delimiter* (or *prompt*) is the boundary between the producer of the intermediate answer (such as the summarization program) and the rest of the system.

Many uses have been discovered for the concept of continuations [1]: in the semantic theory of programming languages [2, 3], as a practical strategy for their design and implementation [4, 5], and in natural-language semantics [6, 7]. In

operating-system research, continuations are poorly known and seldom used explicitly. In this paper, we cross the boundary between operating systems and programming languages to argue by examples that continuations, especially delimited ones, pervade operating systems—if only implicitly. We contend that systems programmers should recognize the applications of delimited continuations, so as to design systems with sensible defaults and implement them using efficient techniques from the literature.

One example of delimited continuations appears in the interaction between an operating system and a user program running under its management. From time to time, the user program may request a service from the kernel of the operating system, for example to read a file. When the kernel receives a request for a system service, it first saves the state, or *execution context*, of the user process. After processing the request, the kernel resumes the process, passing it the reply. If the request takes some time to process, such as when data must be fetched from a hard drive, the operating system may let some other user process run in the meantime and only resume the original user process when the hard drive is done. We can think of the execution context as a function that maps the kernel's reply to the outcome of the user process. This function is a delimited continuation; the delimiter in this case is the boundary between the user process and the rest of the system.

Saving the execution context for a process to be resumed later is called *capturing* the continuation of the process [8]. Usually a captured continuation is invoked exactly once, but sometimes it is invoked multiple times. For example, a typical operating system offers services for a process to duplicate ("fork") itself into two parallel processes or to save a snapshot of itself to be restored in the future. Other times the captured continuation is never invoked, such as when a process invokes the "exit" service to destruct itself. Two or more continuations that invoke each other once each are called *coroutines*. For example, in the PDP-7 Unix operating system, the shell and other user processes transfer control to each other as coroutines (using the "exec" system service [9]).

The concept of an operating-system kernel has found its way into the programming-language literature, for instance to describe in a modular and rigorous way what *side effects* such as state, exceptions, input/output, and backtracking mean [10–12]. A recurring idea in that work is that of a *central authority* [13], mediating interactions between a program, which performs computations, and the external world, which provides resources such as files. A computation yields either a value or a side effect. A side effect is a request to the central authority to perform an action (such as reading a file), paired with a continuation function that accepts the result of the action and resumes the computation.

In practical systems programming, continuations are best known for writing concurrent programs [8, 14–20], distributed programs [21–23], and Web programs [24–29]. In these and many other applications, the programmer codes the handling of events [30] in *continuation-passing style*, whether or not the programmer is aware of the fact. With awareness, continuations have guided the design of a network protocol that does not require the server to track the state of each

connection, and is thus more scalable, easier to migrate, and more resistant to denial-of-service attacks [31].

This paper focuses on a less-known use of continuations: file systems. We stress *transactional* file systems, which treat each operation such as changing, deleting, or renaming a file as a *transaction*, and where a transaction can be undone (that is, rolled back). Our Zipper File System manages each connection between it and its users as a delimited continuation, so it is natural and easy to implement *copy on write*: each user appears to have exclusive use of a separate file system, but the parts that are identical across users are actually stored only once and shared until one user changes its "copy" to be different.

Section 2 gives two examples of delimited continuations in systems programming in more detail. Section 3 describes our Zipper File System. Section 4 reviews the benefits we reap of recognizing continuations explicitly.

2 Instances of continuations

We give two examples of delimited continuations: a user process requesting a system service, and traversing a data structure. The examples seem unrelated, yet use the same programming-language facility (notated `CC` below), thus simplifying their implementation. We have built the Zipper File System as a working prototype of both examples. Our prototype and illustrative code below are written in Haskell, a high-level general-purpose programming language, because it is suitable for operating systems [32] and its notation is concise and close to mathematical specification.

2.1 System calls

The first example is a user process that invokes a system service. As sketched above, the process captures its current continuation and sends it to the kernel along with the requested action. The code below defines a data structure `Req r` that combines the continuation and the action.

```
data Req r = Exit
           | Read (Char -> CC r (Req r))
           | Write Char (() -> CC r (Req r))
```

The possible actions defined are `Exit`, `Read`, and `Write`. An `Exit` request means to destruct the process: it contains no continuation because the process is done. A `Read` request means to read a character: it contains a continuation that accepts the `Char`acter read, yields as the answer another request (usually `Exit`), and may issue more requests during the computation. The type of this continuation, `Char -> CC r (Req r)`, reflects the fact that the continuation may issue more requests: `CC r` marks the type of a computation that may incur side effects, so the type `CC r (Req r)` means a computation that yields `Req r` after possibly incurring side effects. (The parameter `r` is a *region label* [33, 34] and does not concern us here.) A `Write` request means to write a character: it contains the

Character to write, along with a continuation that accepts nothing; hence the type `() -> CC r (Req r)`.

Using these services, we can program a simple user process `cat` to copy the input to the output.

```
service p req = shiftP p (\k -> return (req k))

cat p = do input <- service p Read
           service p (Write input)
           cat p
```

The function `service` initiates a *system call*: `cat` invokes `service` to request reading and writing services from the kernel.

The variable `p` above is a control delimiter: it represents the boundary between the user process and the kernel, delimiting the continuation in a request from the user process to the kernel. In the definition of `service` above, the expression `shiftP p (\k -> ...)` means for the user process to capture the delimited continuation up to the delimiter `p` and call it `k`. Because `p` delimits the user process from the kernel, the delimited continuation `k` is precisely the execution context of the user process. The subexpression `return (req k)` means for the user process to exit to the kernel with a new request data structure containing the captured delimited continuation `k`.

We now turn from how a user process initiates a request to how the operating-system kernel handles the request. The kernel handles system calls in a function called `interpret`, which takes three arguments.

1. The record `world` represents the state of the whole operating system. It includes, among other fields, the *job queue*, a collection of processes waiting to run.
2. The *process control block* `pcb` describes various resources allocated to the current process, such as network connections called *sockets*. Sockets constitute the input and output channels of the process.
3. The request from the process, of type `Req r`, specifies how the process exited along with whether and how it should be resumed.

The function `interpret` is invoked by the *scheduler*, another component of the operating system. The scheduler passes an old `world` to `interpret` and receives in return an updated `world`, then chooses the next process to run from those in the updated job queue.

Let us examine how `interpret` implements `Exit` and `Read` actions. An `Exit` request is handled by disposing of the process' resources, such as by closing its socket. The process itself never resumes, and the memory it uses can be reclaimed right away, because no continuation in the system refers to the process anymore. The process control block can be reclaimed as well.

```
interpret world pcb Exit = do liftIO (sClose (psocket pcb))
                              return world
```

Reading a character may take a long time, and other user processes should be allowed to run in the meantime. Thus the kernel does not respond to a `Read` request immediately. Rather, the `interpret` function creates a record of the pending read on the socket and appends the record to the job queue. It then returns the `world` with the updated job queue to the scheduler.

```
interpret world pcb (Read k) = return world
  {jobQueue = jobQueue world ++ [JQBlockedOnRead pcb k]}
```

The kernel keeps track of the process waiting for a character only by storing the process' continuation in the record of pending read. When the kernel receives data from a socket, it locates any associated read-pending request in the job queue and resumes the blocked process by invoking the function `resume` below.

```
resume world (JQBlockedOnRead pcb k) received_character =
  do req <- k received_character
     interpret world pcb req
```

The function extracts the continuation `k` of the suspended process and passes it the `received_character`, thus resuming the process. The process eventually returns another request `req`, which is `interpret`ed as above.

This example shows how a process that just yielded control (to the kernel) is a continuation [14]. We have in fact implemented delimited continuations in the Perl 5 programming language by representing them as server processes that yield control until they receive a client connection. Although the mathematical meaning of a delimited continuation is a function that maps request values from a client to response answers from the server, the function is represented by data structures [35] and so can be saved into a file or sent to remote hosts. To save a captured continuation to be reused later is to take a snapshot of a process, or to *checkpoint* it.

The control delimiter `p` in the code above delimits the kernel from a user process. The same kind of delimiters can be used by a user process such as a debugger to run a subcomputation in a sandbox and intercept requests from the sandbox before forwarding them to the kernel. This *interposition* facility falls out from our view of requests as containing delimited continuations.

2.2 Data traversal

The second, seemingly unrelated example of delimited continuations is the traversal and update of a complex data structure. For simplicity, instead of a directory tree, we consider here a binary tree in which each node either contains two branches or is a leaf node labeled by an integer.

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

We define an operation to traverse the leaves of the tree, perhaps returning a new, updated version for some of them.

```
traverse :: Monad m => (Int -> m (Maybe Int)) -> Tree -> m Tree
traverse visit l@(Leaf n) = do result <- visit n
                               return (maybe l Leaf result)
traverse visit (Node l r) = do l <- traverse visit l
                               r <- traverse visit r
                               return (Node l r)
```

The first argument to the `traverse` function, `visit`, is itself a function, of type `Int -> m (Maybe Int)`. It takes the integer label of the current leaf node and returns either `Nothing` or a new label with which to update the node. For example, the following code makes a tree like `tree1` except all leaf labels less than 2 are replaced with 5.

```
traverse (\n -> return (if n < 2 then Just 5 else Nothing)) tree1
```

The update is *nondestructive*: the old `tree1` is intact and may be regarded as a snapshot of the data before the update. If `tree1` is not used further in the computation, the system will reclaim the storage space it occupies. To use `tree1` further, on the other hand, is to "undo" the update. The nondestructive update takes little more memory than a destructive update would, because the new tree *shares* any unmodified data with the old tree. That is, `traverse` performs copy-on-write. (The code above actually only shares unmodified *leaves* among traversals. A slight modification of the code, implemented in the Zipper File System, lets us share unmodified *branches* as well.)

Another benefit of the nondestructive update performed by `traverse` is *isolation*: any other computation using `tree1` at the same time will be unaffected by our update and may proceed concurrently. Two concurrent traversals that wish to know of each other's updates must exchange them, possibly through a common arbiter—the operating-system kernel—using the same system-call interface based on delimited continuations discussed in Section 2.1. The arbiter may reconcile or reject the updates and report the result to the concurrent traversals. The outcome does not depend on the order in which the updates are performed—that is, we avoid *race conditions*—because nondestructive updates do not modify the same original version of the data that they share. Nondestructive updates of the same sort are used in distributed revision control and in robust distributed telecom software [36].

For reading and updating a file, file system, process tree, or database, an interface like `traverse` is a more appropriate access primitive than the *cursor*-based (or *handle*-based) interface more prevalent today, in that the traversal interface eliminates the risk of forgetting to dispose of a cursor or trying to use a cursor already disposed of [37]. The traversal interface is no less expressive: when the cursor-based access is truly required, it can be automatically obtained from the traversal interface using delimited continuations, as we now explain.

The *zipper* [38] data-type `Z r` is what is commonly called a database cursor or file handle.

```
data Z r = Done Tree | Yet Int (Maybe Int -> CC r (Z r))
```

A zipper's state is either `Done` or `Yet`. A `Done` zipper has finished traversing the old tree and holds a new tree. A `Yet` zipper represents an unfinished traversal and holds the current leaf label (`Int`) and a continuation to advance the traversal (`Maybe Int -> CC r (Z r)`).

The zipper provides the following interface. The `open` function begins a traversal on an initial tree. The `curr` function reads the current leaf label. The `next` function advances the traversal, whereas `write` updates the current leaf label then advances the traversal. The `close` function finishes the traversal and returns the new tree.

```
open :: Tree -> CC r (Z r)
open tree = promptP (\p -> let visit n = shiftP p (return . Yet n)
                           in liftM Done (traverse visit tree))

curr :: Z r -> Int
curr (Yet n _) = n

next :: Z r -> CC r (Z r)
next (Yet _ k) = k Nothing

write :: Int -> Z r -> CC r (Z r)
write n (Yet _ k) = k (Just n)

close :: Z r -> CC r Tree
close (Done tree) = return tree
close z = next z >>= close
```

The sample program below uses these functions to add the first leaf label to the second leaf label.

```
test2 = runCC (do z1 <- open tree1
                  let s1 = curr z1
                  z2 <- next z1
                  let s2 = curr z2
                  z3 <- write (s1+s2) z2
                  close z3)
```

This programming style is like using a database cursor or file handle, except the functions `next` and `write` are nondestructive and return new zippers (`z2` and `z3` above) to reflect the new state of the tree. Using the old zippers (`z1` and `z2`), we can recall any past state of the traversal, undoing the updates after that point. If we do not use the old zippers, the system will reclaim the storage space they occupy. As before, different zippers from the same traversal share data by copy-on-write. To save a captured continuation to be reused later is to take a *snapshot* of the data.

3 The Zipper File System

The Zipper File System is a prototype file server and operating system. It consists of only about 1000 lines of Haskell code, about half of which implements delimited continuations and zippers. It provides multitasking, exception handling,

and transactional storage all using delimited continuations. More information, including complete source code, is available online at `http://okmij.org/ftp/Computation/Continuations.html#zipper-fs`

Storage in the Zipper File System is a data structure much like the `Tree` above, except leaves contain file data and tree nodes have an arbitrary number of branches, identified by string names that serve the same role as directory and file names in a conventional file system. The system exports the traversal and zipper operations described above as an interface for client access. A simple kernel manages *shell* processes, each of which lets a user access this interface over a network connection. Multiple users may connect at the same time and use commands such as `ls` (list directory contents), `cat` (display directory contents), `cd` (work in another directory), `touch` (create a file), `mkdir` (create a directory), `rm` (delete), `mv` (move), `cp` (copy), and `echo` (write a literal string to a file). Thanks to the copy-on-write semantics that arises naturally from the use of delimited continuations, the `cp` (copy) command need only establish sharing between two locations in the file system, not copy any actual file data. Unlike in the Unix operating system, one can traverse sequentially to the *next* node from any node.

The kernel uses delimited continuations to provide system calls and schedule which user process to run next. The *type system* isolates the processes from each other and prevents them from performing input/output or changing global state except by issuing a request to the kernel. Thus any processor can potentially be scheduled to run any process without worrying about the undesirable interactions that often result when two processes access the same memory at the same time. This protection is similar to that among Unix processes, except we enforce it by programming-language types in software rather than a memory-management unit in hardware.

For a user of the Zipper File System, what most sets it apart is the transactional semantics of its storage. The user can undo mistakes such as deleting a directory or truncating a file. Moreover, multiple users are completely isolated from each other: each network connection appears to expose exclusive use of a separate file system, as if every operation always occurs before or after every other operation, never concurrently. Data unmodified by two clients are shared across them. These features all come for free with the zipper.

As with database transactions, a client may announce its update by "committing" it. The commit request is handled by a central authority, which examines the update and accepts or rejects it, with no risk of race conditions. Any transaction system needs a conflict resolution mechanism such as versioning, patching, or manual intervention. Our system resolves conflicts in the central authority that maintains the global state, rather than in user processes, which cannot change the global state directly. The conflict-resolution policies are thus easier to implement.

4 Conclusion

We have described how the Zipper File System explicitly uses delimited continuations for multitasking and storage. For storage, delimited continuations make it natural and easy to provide a transactional semantics, complete isolation, and sequential traversal. For multitasking, delimited continuations make it natural and easy to schedule processes for execution, respond to input and output events, and handle exceptions. In both applications, delimited continuations avail us of the state of the art in implementation techniques, such as copy-on-write and *stack segmentation* [39–41].

The recent surge of operating systems and file systems implemented in high-level programming languages [32, 42, 43] find their roots in earlier systems such as Multics, Inferno, and SPIN. Our work shows how delimited continuations are particularly helpful, especially in conjunction with types that describe the shape of data and effect of code in detail. We use such types to sandbox processes, isolate transactions, prevent race conditions, improve scalability to multiple processors, and obviate the user-kernel boundary in hardware.

We treat the file system, which is usually thought of as a persistent data structure, as an ongoing traversal process that communicates with the outside world as a coroutine. More generally, data as small as a single integer variable can be profitably treated as a process with which to exchange messages [44, 45]. These alternating viewpoints between process and data prompt us to ask: could the vision of persistent virtual memory pioneered by Multics be relevant in today's world of ubiquitous memory management units?

Any software component can interact with the rest of the world using delimited continuations. When the continuations are isolated by restrictions on side effects, the interaction naturally and easily supports snapshots, undo, and reconciliation. Thus to use an operating system can and should be to navigate a virtual file system containing the history and transcript of all potential interactions.

References

1. Reynolds, J.C.: The discoveries of continuations. Lisp and Symbolic Computation **6** (1993) 233–247
2. Strachey, C., Wadsworth, C.P.: Continuations: A mathematical semantics for handling full jumps. Higher-Order and Symbolic Computation **13** (2000) 135–152
3. Fischer, M.J.: Lambda-calculus schemata. Lisp and Symbolic Computation **6** (1993) 259–288
4. Kelsey, R., Clinger, W.D., Rees, J., Abelson, H., Dybvig, R.K., Haynes, C.T., Rozas, G.J., Adams, IV, N.I., Friedman, D.P., Kohlbecker, E., Steele, G.L., Bartley, D.H., Halstead, R., Oxley, D., Sussman, G.J., Brooks, G., Hanson, C., Pitman, K.M., Wand, M.: Revised^5 report on the algorithmic language Scheme. Higher-Order and Symbolic Computation **11** (1998) 7–105 Also as *ACM SIGPLAN Notices* 33(9):26–76.
5. Steele, Jr., G.L.: RABBIT: A compiler for SCHEME. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (1978) Also as Memo 474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

6. Shan, C.c., Barker, C.: Explaining crossover and superiority as left-to-right evaluation. Linguistics and Philosophy **29** (2006) 91–134
7. Barker, C., Shan, C.c.: Types as graphs: Continuations in type logical grammar. Journal of Logic, Language and Information **15** (2006) 331–370
8. Wand, M.: Continuation-based multiprocessing revisited. Higher-Order and Symbolic Computation (1999) 283
9. Ritchie, D.M.: The Evolution of the Unix Time-sharing System. AT&T Bell Laboratories Technical Journal **63** (1984) 1577–93
10. Sitaram, D., Felleisen, M.: Control delimiters and their hierarchies. Lisp and Symbolic Computation **3** (1990) 67–99
11. Kiselyov, O.: How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Computer Science Department, Indiana University (2005)
12. Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers (functional pearl). In: ICFP '05: Proceedings of the ACM International Conference on Functional Programming, ACM Press (2005) 192–203
13. Cartwright, R., Felleisen, M.: Extensible denotational language specifications. In Hagiya, M., Mitchell, J.C., eds.: Theoretical Aspects of Computer Software: International Symposium. Number 789 in Lecture Notes in Computer Science, Springer-Verlag (1994) 244–272
14. Kumar, S., Bruggeman, C., Dybvig, R.K.: Threads yield continuations. Lisp and Symbolic Computation **10, 2** (1998) 223–236
15. Dybvig, R.K., Hieb, R.: Continuations and concurrency. In: Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (1990) 128–136
16. Dybvig, R.K., Hieb, R.: Engines from continuations. Journal of Computer Languages **14, 2** (1989) 109–123
17. Shivers, O.: Continuations and threads: Expressing machine concurrency directly in advanced languages. In: Proceedings of the Second ACM SIGPLAN Workshop on Continuations. (1997)
18. Haynes, C.T., Friedman, D.P., Wand, M.: Obtaining coroutines with continuations. Journal of Computer Languages **11** (1986) 143–153
19. Li, P., Zdancewic, S.: A language-based approach to unifying events and threads. `http://www.cis.upenn.edu/~stevez/papers/LZ06b.pdf` (2006)
20. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management, or, event-driven programming is not the opposite of threaded programming. In: Proceedings of the 2002 USENIX Annual Technical Conference, USENIX (2002) 289–302
21. Sumii, E.: An implementation of transparent migration on standard Scheme. In Felleisen, M., ed.: Proceedings of the Workshop on Scheme and Functional Programming. Number 00-368 in Tech. Rep., Department of Computer Science, Rice University (2000) 61–63
22. Sewell, P., Leifer, J.J., Wansbrough, K., Zappa Nardelli, F., Allen-Williams, M., Habouzit, P., Vafeiadis, V.: Acute: High-level programming language design for distributed computation. In: ICFP '05: Proceedings of the ACM International Conference on Functional Programming, ACM Press (2005) 15–26
23. Murphy, VII, T., Crary, K., Harper, R.: Distributed control flow with classical modal logic. In Ong, C.H.L., ed.: Computer Science Logic: 19th International Workshop, CSL 2005. Number 3634 in Lecture Notes in Computer Science, Springer-Verlag (2005) 51–69

24. Queinnec, C.: Continuations and web servers. Higher-Order and Symbolic Computation **17** (2004) 277–295
25. Graunke, P.T.: Web Interactions. PhD thesis, College of Computer Science, Northeastern University (2003)
26. Colomba, A.: SISCweb: A framework to facilitate writing stateful Scheme web applications in a J2EE environment. `http://siscweb.sf.net/` (2007)
27. Balat, V., et al.: Ocsigen: A Web server and a programming framework providing a new way to create dynamic Web sites. `http://www.ocsigen.org` (2007)
28. Belapurkar, A.: Use continuations to develop complex Web applications. IBM developerWorks (2004)
29. Krishnamurthi, S., Hopkins, P.W., McCarthy, J., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Implementation and use of the PLT Scheme Web server. Higher-Order and Symbolic Computation (2007)
30. Williams, N.J.: An implementation of scheduler activations on the NetBSD operating system. In: Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, Berkeley, CA, USENIX (2002) 99–108
31. Shieh, A., Myers, A., Sirer, E.G.: Trickles: A stateless transport protocol. Summaries of OSDI'04. USENIX ;login: vol. 30, No. 2, 2005, p. 66 (2004) 6th Symposium on Operating Systems Design and Implementation, OSDI'04. Work-in-Progress Reports.
32. Hallgren, T., Jones, M.P., Leslie, R., Tolmach, A.P.: A principled approach to operating system construction in Haskell. In Danvy, O., Pierce, B.C., eds.: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005, ACM (2005) 116–128
33. Launchbury, J., Peyton Jones, S.L.: State in Haskell. Lisp and Symbolic Computation **8** (1995) 293–341
34. Moggi, E., Sabry, A.: Monadic encapsulation of effects: A revised approach (extended version). Journal of Functional Programming **11** (2001) 591–627
35. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the 3rd International Conference on Principles and Practice of Declarative Programming, ACM Press (2001) 162–174
36. Nyström, J.H., Trinder, P.W., King, D.J.: Are high-level languages suitable for robust telecoms software? In Winther, R., Gran, B.A., Dahll, G., eds.: Computer Safety, Reliability, and Security, 24th International Conference, SAFECOMP 2005, Fredrikstad, Norway, September 28-30, 2005, Proceedings. Volume 3688 of Lecture Notes in Computer Science., Springer (2005) 275–288
37. Kiselyov, O.: General ways to traverse collections. `http://okmij.org/ftp/Scheme/enumerators-callcc.html`; `http://okmij.org/ftp/Computation/Continuations.html` (2004)
38. Huet, G.: The zipper. Journal of Functional Programming **7** (1997) 549–554
39. Clinger, W.D., Hartheimer, A., Ost, E.M.: Implementation strategies for continuations. Higher-Order and Symbolic Computation **Vol. 12** (1999) 7–45
40. Bruggeman, C., Waddell, O., Dybvig, R.K.: Representing control in the presence of one-shot continuations. In: ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation. (1996)
41. Gasbichler, M., Sperber, M.: Final shift for call/cc: Direct implementation of shift and reset. In: ICFP '02: Proceedings of the ACM International Conference on Functional Programming, ACM Press (2002) 271–282
42. Derrin, P., Elphinstone, K., Klein, G., Cock, D., Chakravarty, M.M.T.: Running the manual: an approach to high-assurance microkernel development. In: Haskell

'06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, ACM Press (2006) 60–71
43. Jones, I., et al.: Halfs, a Haskell filesystem. `http://www.haskell.org/halfs/` (2006)
44. Ernst, E.: Method mixins. Report PB-557, Department of Computer Science, University of Aarhus, Denmark (2002)
45. Van Roy, P.: Convergence in language design: A case of lightning striking four times in the same place. In Hagiya, M., Wadler, P., eds.: Proceedings of FLOPS 2006: 8th International Symposium on Functional and Logic Programming. Number 3945 in Lecture Notes in Computer Science, Springer-Verlag (2006) 2–12