Reuse of Continuation-based Control-flow Abstractions

Steven te Brinke, Christoph Bockisch, and Lodewijk Bergmans
University of Twente – Software Engineering group – Enschede, The Netherlands
{brinkes, c.m.bockisch, bergmans}@cs.utwente.nl

ABSTRACT

It is commonly agreed that decomposing a software design according to the structure of the problem domain makes it easier to maintain and manage its complexity (e.g. [9]). To retain the resulting decomposition in the implementation, a programming language is needed that supports composition mechanisms that match the compositions in the problem domain. However, current programming languages only offer a small, fixed set of composition mechanisms. This leads to compromised implementations that either do not follow the decomposition from the design, or are cluttered with additional glue code. In this paper, we discuss the composition of control flow and investigate the hypothesis that continuations may be a suitable atomic core principle for offering a wide range of control-flow composition mechanisms in a form that does not compromise the structure and maintainability of the application. We present four variants of exception handling, which have been implemented using continuations and discuss their differences and commonalities. The results of this comparison are encouraging with respect to the expressiveness of continuations and reusability of implementations. But they also show that current language support for continuations leads to code replication and complex code.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

1. INTRODUCTION

Software composition is a key element in software engineering and programming language technology: it is the primary means for managing the size and complexity of large software construction projects [9]. A *composition mechanism* lets programmers define—part of—the behavior of an abstraction, in terms of the behavior of one or more other abstractions. Previously [3, 12] we—as well as others [18, 6]—have argued that fixing the composition mechanisms offered by a language severely limits developers.

To be able to choose the most appropriate decomposition, developers should not be restricted by the programming language to just a few composition mechanisms. Such restrictions require workarounds leading to code duplication and obfuscation of the actual intention of the composition. Our general approach is to enable the implementation of composition mechanisms as first-class objects in the programming language. With such an approach, composition mechanisms can be distributed as libraries, allowing developers to choose freely. Furthermore, developers can derive new composition mechanisms if their domain-specific decomposition requires a specific variant; new mechanisms can be added without changing the actual programming language.

In our work, we consider composition in a broad sense; from large-grained composition mechanisms such as inheritance and aggregation of classes or components, to composition of procedures by means of procedure calls, down to fine-grained composition of groups of program elements by e.g. control structures. A composition mechanism may involve several semantic concerns; previously, we have explored behavior-related concerns, expressing different behavioral compositions such as inheritance, delegation, and aspects [12], and we explored data-related concerns, expressing varying semantics for e.g. data sharing and visibility [5]. In this paper, we focus on the control flow related semantics of compositions. Typically, this refers to the control flow of method invocations, data access or expression evaluation. We are particularly interested in control-flow composition going beyond a simple request-reply model of method invocation like exception handling, co-routines, or around advice with proceed in aspect-oriented programming.

Continuations are a programming language feature that bundles a set of instructions together with an execution state into a first-class entity. They are referred to as a generalization of *goto*. In this paper, we investigate the hypothesis that continuations may be a suitable atomic core principle that is able to express all, or many, of the well-known variations in handling control flow. Our eventual interest is in adopting continuations as a part of a generic composition model, able to express a wide variation of composition techniques through a single principled model dealing with control-flow composition *and* behavior composition.

There is a lot of literature that shows how existing language features for managing control flow can be implemented with continuations. In this paper we focus on investigat-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FREECO-Onward! 2011, October 23, 2011, Portland, Oregon, USA.
Copyright 2011 ACM 978-1-4503-1025-3/11/10 ...$10.00.

```
1  def log(message) = {
2    var logger = null
3    try {
4      logger = new Logger("debug.log")
5      logger.log(message)
6    } catch { case e =>
7
8      println("Error while logging: " + e.getMessage())
9
10     throw e
11   } finally {
12     if (logger != null) {
13
14       logger.close()
15
16     }
17   }
18 }
```

Listing 1: Using try catch finally in Scala

```
1  def log(message, ret, thro) = {
2    var logger = null
3    tryCatchFinally({ (ret, thro) =>
4      logger = new Logger("debug.log")
5      logger.log(message, ret, thro)
6    }, { (e, ret, thro) =>
7      callcc {
8        println("Error while logging: " + e.getMessage(), _, thro)
9      }
10     thro (e)
11   }, { (ret, thro) =>
12     if (logger != null) {
13       callcc {
14         logger.close(_, thro)
15       }
16     }
17   }, ret, thro)
18 }
```

Listing 2: Using try catch finally with continuations

ing the reusability of continuation-based implementations of control-flow composition mechanisms. We do this by discussing the continuation-based implementation in Scala of four variants of exception handling. The implementations are embodied in functions, for which Scala already offers reuse mechanisms. The main novelty of this paper consists in the discussion of differences in the variant implementations. We conclude that continuations are suitable to reusably implement different variants of control-flow composition; but new language features are required to compose and refine implementations of such mechanisms to avoid redundancy. Besides exception handling, which is presented in detail, we discuss a variety of additional control-flow composition mechanisms, of which several variants exist, and which can also be implemented with continuations.

2. CONTROL-FLOW COMPOSITION WITH CONTINUATIONS

First-class continuations provide the ability to store an execution state (this is, e.g., the call stack together with an instruction pointer) and to continue execution in this state at a later point in the program [7]. The control operator *call-with-current-continuation* (callcc) invokes a function f and at the same time reifies the *current continuation*, which is the execution state at the time when the invocation of f is performed, where the instruction pointer refers to the instruction right after the invocation. This control operator implicitly passes the current continuation to f. *Continuation-passing style* (CPS) is a programming style where no function call ever returns. Instead, the rest of the computation to be performed after the function finished must be passed as a continuation.

The continuation-passing style is often referred to as a mechanism suitable to implement many control-flow-related language concepts, for example co-routines, loops and exceptions. To our knowledge, existing publications only present the implementation of one variant of a specific mechanism. In this paper, we want to point out the commonalities in control-flow composition mechanisms and the potential for reuse in their continuation-based implementations.

As a case study, we present implementations of different exception handling mechanisms and discuss their commonalities and differences. The examples will be presented in Scala [15] and are tested with Scala's semantics for continuations and context binding in continuations.[1] Scala is an object-oriented programming language with support for so-called delimited continuations [17]. This allows programmers to mix normal and CPS code.

3. EXCEPTION HANDLING

To start with, consider the Scala code in listing 1 that uses Scala's built-in exception handling mechanism. The operations within the **try**{} block may throw an exception—e.g., when the disk is full—which is in turn handled by the operations within the **catch**{} block; the operations within the **finally**{} block are always executed regardless whether the **try**{} block terminated normally or exceptionally.

3.1 Exception handling with continuations

Listing 2 shows the implementation of the same behavior with continuations. The main difference is that function calls are wrapped in a callcc block. As example, consider the call to println on lines 7–9; the current continuation is passed to println at the argument position denoted by '_'.

The structure of listing 2 is the same as that of listing 1: we see a call to tryCatchFinally with continuations as arguments representing the **try**, **catch**, and **finally** blocks. To emphasize this correspondence and to simplify the discussion below, we also call these continuations "blocks".

The implementation of the semantics of composing control flow in the presence of exceptions is embodied in the function tryCatchFinally (shown in listing 3) but also in the expected signature of the continuations passed to this function. The five arguments of tryCatchFinally represent the following:

1. The try block. This continuation must accepts two arguments to be passed when the try block is executed:
   (a) The so-called *return continuation* to be executed when the try block terminates normally.
   (b) The so-called *throw continuation* that can be invoked to throw an exception.
2. The catch block. This continuation is called if an exception is thrown in the try block; its arguments are:
   (a) The raised exception.
   (b,c) The return and throw continuations, which have the same role as the arguments of the try block.
3. The finally block. This continuation is called when

[1]The showed examples are shortened, their full versions can be downloaded from `http://co-op.sf.net`.

```
1  def tryCatchFinally(dotry, docatch, dofinally, ret, thro) {
2    var finallyPlain = { () =>
3      dofinally(ret, thro)
4    }
5    var finallyRethrow = { e : Throwable =>
6      finallyPlain()
7      thro(e)
8    }
9    var finallyRereturn = { result =>
10     finallyPlain()
11     ret(result)
12   }
13   var catchPlain = { e : Throwable =>
14     docatch(e, finallyRereturn, finallyRethrow)
15     finallyPlain()
16   }
17   var tryPlain = {
18     dotry(finallyRereturn, catchPlain)
19     finallyPlain()
20   }
21   tryPlain
22 }
```

Listing 3: Defining try catch finally with continuations

the catch block finishes or when the try block finishes normally. It takes the same arguments as the try block.

4. The continuation to be executed when the try-catch-finally block terminates normally.
5. The continuation to be invoked when an exception thrown in the catch or finally blocks must be passed on to an enclosing exception handler.

In short, the implementation of tryCatchFinally works as follows. When we look at listing 2, the *try*, *catch* and *finally* blocks all terminate in one of three ways. They either (1) return from their defining method (i.e. call the continuation ret), in this case log, (2) throw an exception (i.e. call the continuation thro) or (3) 'fall off' (finish without performing any of the former two behaviors). The actual behavior of these three situations is defined in listing 3 by the continuations passed to the blocks. When the *try* block falls off, only the continuation finallyPlain is executed. When the *try* or *catch* blocks return, they invoke finallyRereturn, which executes the continuation finallyPlain before actually returning using ret. Since ret (line 11) never returns, dofinally is executed only once (line 19 is never reached in this case). When the *try* block throws an exception, it invokes the continuation catchPlain. When the *catch* block throws an exception, it invokes finallyRethrow, which executes the finallyPlain continuation before throwing the exception using thro. This implementation is inspired by an example presented by Might.[2]

3.2 Multi-catch exception handling

From languages such as Java, developers are used to provide multiple catch blocks with a declarative definition of which type is handled by each of them. Listing 4 shows a modified catch block that uses this ability. It consists of a list of pairs specifying for which exception types (first element) the handler (second element) should be triggered.

Listing 5 shows a modified version of our try-catch-finally implementation that allows using the catch blocks presented in listing 4. A box encloses the code which is different compared to the original version. The modification changes the handling of the exceptional case only. The catchPlain continuation invokes the first handler that can handle the thrown exception (i.e. the thrown exception is assignment compatible with the type specified by the handler).

[2]See his blog post at `http://matt.might.net/articles/implementing-exceptions/`.

```
1  List(
2    (classOf[FileNotFoundException], // Exception type
3      (e, ret, thro) => { // Handler
4        callcc {
5          println("Invalid log filename: " + e.getMessage(), _, thro)
6        }
7        thro (e)
8      }
9    ),
10   (classOf[IOException], // Exception type
11     (e, ret, thro) => { // Handler
12       callcc {
13         println("Error while logging: " + e.getMessage(), _, thro)
14       }
15       thro (e)
16     }
17   )
18 )
```

Listing 4: Exceptions with multiple catch blocks

```
1  def tryCatchFinally(dotry, handlers, dofinally, ret, thro) {
2    var finallyPlain = { () =>
3      dofinally(ret, thro)
4    }
5    var finallyRethrow = { e : Throwable =>
6      finallyPlain()
7      thro(e)
8    }
9    var finallyRereturn = { result =>
10     finallyPlain()
11     ret(result)
12   }
13   var catchPlain = { e : Throwable =>
14       handlers.find(h => h._1.isInstance(e)) match {
15         case Some((_, docatch)) =>
16           docatch(e, finallyRereturn, finallyRethrow)
17           finallyPlain()
18         case None =>
19           finallyPlain()
20           thro(e) // No handler, so throw on
21       }
22   }
23   var tryPlain = {
24     dotry(finallyRereturn, catchPlain)
25     finallyPlain()
26   }
27   tryPlain
28 }
```

Listing 5: Defining try catch finally, allowing multiple catch blocks

3.3 Retrying and resuming exceptions

Two more variants of exception handling are retrying [14] and resuming exceptions. With retrying exceptions, a catch block can re-execute the try block. For example when the disk is full, an I/O exception will be thrown. The catch block could show a dialog asking the user to free disk space and retry writing to the file afterward. With resuming exceptions, a catch block can resume the execution after the instruction throwing the exception. As example consider an I/O exception while writing to a file which is not vital; a catch block may try to recover from that, but may also decide to ignore this exception and skip the I/O operation.

```
1  { (e, ret, thro, retry) =>
2      callcc {
3        println("Error while logging: " + e.getMessage(), _, thro)
4      }
5      ... // Allow user to fix cause of failure, e.g. by freeing disk space
```

```
6       retry()
7   }
```

Listing 6: Catch block using retrying exceptions

Listing 6 shows the use of retrying exceptions, of which an implementation is given in listing 7. Again, boxes enclose the differences compared to the original version shown in listing 3, which lie only in the signatures of catch and throw.

```
1   def tryCatchFinally(dotry, docatch, dofinally, ret, thro) {
2     var finallyPlain = { () =>
3       dofinally(ret, thro)
4     }
5     var finallyRethrow = { (e : Throwable, retry) =>
6       finallyPlain()
7       thro(e, retry)
8     }
9     var finallyReturn = { result =>
10      finallyPlain()
11      ret(result)
12    }
13    var catchPlain = { (e : Throwable, retry) =>
14      docatch(e, finallyReturn, finallyRethrow, retry)
15      finallyPlain()
16    }
17    var tryPlain = {
18      dotry(finallyReturn, catchPlain)
19      finallyPlain()
20    }
21    tryPlain
22  }
```

Listing 7: Defining try catch finally, allowing catch blocks to retry

Listing 8 shows the use of resuming exceptions. The try-catch-finally implementation for resuming exceptions is equivalent to that for retrying exceptions (shown in listing 7). The difference is their usage: for retrying exceptions, the second argument of the throw continuation represents the start of the try block of the failing operation, whereas with resumable exceptions, it represents the rest of the try block's computation after the excepting instruction.

```
1   { (e, ret, thro, ignore) =>
2     callcc {
3       println("Error while logging: " + e.getMessage(), _, thro)
4     }
5     ignore()
6   }
```

Listing 8: Catch block using resuming exceptions

As variant, resuming exceptions can be augmented by registering an exception handler higher up the call chain, which, e.g., ignores all I/O exceptions. This way, failing I/O will be ignored whenever it is not handled by any other exception handler. Another variant is multi-catch exception handling (shown in section 3.2) where the most specific handler is executed instead of the first matching.

4. LESSONS LEARNED

As can be seen in the examples above, different variants of control-flow composition mechanisms share the largest part of their implementation; at the same time, we have shown that continuations are a suitable mechanism for modeling different control-flow composition mechanisms. By implementing the semantics of such mechanisms in terms of a function, it is possible to provide the implementation as library and developers can choose to use any of the variants, even multiple variants in the same project.

To make this approach actually feasible, we see two points that need improvement. First, reusing control-flow composition mechanisms as a whole is already possible as outlined above. But to get rid of the redundancies we observed in the implementations, means must be developed to also refine such mechanisms and possibly compose composition mechanisms themselves. It should be possible to implement the variants like multi-catch-block, retrying or resuming exceptions as extensions of the original try-catch-finally implementation without any code redundancy. Ideally, it should also be possible to compose several variants easily to, e.g., define exceptions which can both be retried and resumed.

Second, we would like to hide the complexity of programming in continuation-passing style while still being able to use its ability to modularize the implementation of control-flow composition mechanisms.

5. ADDITIONAL EXAMPLES

We implemented four kinds of exception handling[3] and showed that more variants exist. Additional examples of control-flow composition mechanisms which can be modelled with continuations are:

- control statements, e.g. for, while, do, if and switch,
- coroutines and generators,
- concurrency,
- actors,
- reactive programming,
- asynchronous I/O and futures,
- monads,
- lazy evaluation,
- iterators, selection and projection,
- around advice including proceed.

For most of these composition mechanisms, a continuation based implementation is already given by Might[4], Rompf et al. [17], Wang [20], or Filinski [10]. However, the similarities between the different approaches is not explored. We expect that different control structures will be overlapping in similar ways as exception handling, and also have variants which are not listed here. Therefore, we expect that these composition mechanisms have similar reuse opportunities as exception handling does.

6. RELATED WORK

Publications discussing the implementation of control-flow abstractions with continuations have been discussed throughout this paper. Additional documentation [1, 13] presents approaches for compiling control-flow statements to continuation-passing style. To the best of our knowledge, the *comparison* of control-flow abstractions based on a common implementation technique, however, is missing in literature.

Metaspin [4] uses continuation for implementing around advices, but does not express these as first-class entities that can be accessed by the programmer.

Wakita [19] uses two variants of first-class continuations: method and message continuations. Method continuations are first class continuations similar to the ones we use, message continuations are used to create first-class messages.

[3]Three of these types of exception handling are described at `http://c2.com/cgi/wiki?AbortRetryIgnore`.

[4]See his blog posts at `http://matt.might.net/articles/`, section Continuations.

However, continuations are only discussed from an implementation perspective; there is no discussion of their usage by application programmers.

In Pharo [2, chapter Handling exceptions], a Smalltalk environment, different kinds of exceptions are implemented uniformly. However, their implementation does not use a generic language mechanism suitable for implementing other abstractions, but relies on VM support that is specific to exception handling.

To overcome the complexity brought by CPS, C# 5 introduces the keywords *async* and *await* to achieve asynchronous computations.[5] For ECMAScript extended with generators, which adds the keyword *yield* to the language, the same is achieved as library functions.[6]

Several unified implementation techniques are discussed in literature, which, however, are limited to modeling a request-reply-style of composition rather than arbitrary control flow composition as presented in this paper. As examples consider the work on predicate dispatching by Ernst et al. [8] or on associative lookup by Piumarta [16].

7. CONCLUSIONS AND FUTURE WORK

From our literature study and our own experience while implementing the examples in this paper, we conclude that continuations are sufficiently powerful to implement the composition of control flow abstractions. Since the semantics of a composition mechanism is implemented in functions and since there are well-established mechanisms for reusing function definitions, a continuation-based implementation furthermore fulfills our reusability requirement.

In our case study we recognized that there is large overlap between implementations of different control-flow composition variants. Therefore, a way of refining continuation-based implementations of control-flow composition mechanisms is needed that avoids code replication. Furthermore, the continuation-passing style puts a burden on the application developer as the code becomes more complicated than in a programming style where functions always return.

We expect that these issues can be improved by designing a programming language offering mechanisms to define and handle continuation-based implementations of composition mechanisms. For this goal, we will research new abstractions for defining continuations. For example, whether or not the current continuation is passed to a function or continuation implementing a composition mechanism, determines its semantics. This has been shown by the example of retrying and resuming exceptions which only differs in which continuation is passed to the throwing continuation. Thus, a function should declare whether it is called with the current continuation or not, rather than requiring that invocations are in a callcc block.

8. REFERENCES

[1] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Symposium on Principles of Programming Languages*, pages 293–302, 1989.

[2] A. Bergel, D. Cassou, S. Ducasse, and J. Laval. *More Pharo by Example*. 2nd edition, 2011. Draft, `http://pharobyexample.org`.

[3] L. M. J. Bergmans, C. M. Bockisch, and M. Akşit. Liberating composition from language dictatorship. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2011.

[4] J. Brichau, M. Mezini, J. Noyé, W. K. Havinga, L. M. J. Bergmans, V. Gasiunas, C. M. Bockisch, J. Fabry, and T. D'Hondt. An initial metamodel for aspect-oriented languages. Technical report, AOSD-Europe, Feb. 2006.

[5] S. te Brinke, L. M. J. Bergmans, and C. M. Bockisch. The keyword revolution: promoting language constructs for data access to first class citizens. In FREECO '11 [11], pages 4:1–4:5.

[6] F. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 1987.

[7] O. Danvy. On evaluation contexts, continuations, and the rest of the computation. In H. Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW'04)*, number CSR-04-1, Birmingham B15 2TT, United Kingdom, Jan. 2004.

[8] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. *Lecture Notes in Computer Science*, 1445:186–211, 1998.

[9] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[10] A. Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 446–457, New York, NY, USA, 1994. ACM.

[11] *Proceedings of the 1st International Workshop on Free Composition*, FREECO '11, New York, NY, USA, July 2011. ACM.

[12] W. K. Havinga, L. M. J. Bergmans, and M. Akşit. A model for composable composition operators: Expressing object and aspect compositions with first-class operators. In *Proceedings of AOSD*, pages 145–156. ACM, March 2010.

[13] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: an optimizing compiler for scheme. *SIGPLAN Not.*, 21:219–233, July 1986.

[14] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[15] M. Odersky. *The Scala Language Specification, Version 2.9*. Programming Methods Laboratory, May 2011.

[16] I. Piumarta. An association-based model of dynamic behaviour. In FREECO '11 [11], pages 3:1–3:5.

[17] T. Rompf, I. Maier, and M. Odersky. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 317–328, New York, NY, USA, 2009. ACM.

[18] G. L. Steele Jr. Growing a language. *Higher Order Symbol. Comput.*, 12(3):221–236, 1999.

[19] K. Wakita. First class messages as first class continuations. In *Proceedings of the International Symposium on Object Technologies for Advanced Software*, pages 442–459, 1993.

[20] C. Wang. Obtaining lazy evaluation with continuations in scheme. *Information Processing Letters*, 35:93–97, 1990.

[5] See Eric Lippert's blog posts at `http://blogs.msdn.com/b/ericlippert/archive/tags/async/`.

[6] See the ECMAScript wiki: `http://wiki.ecmascript.org/doku.php?id=strawman:async_functions`.