An Advice Mechanism for Non-local Flow Control

Hidehiko Masuhara   Kenta Fujita   Tomoyuki Aotani
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology, Japan
masuhara@acm.org   fujita.k.ak@m.titech.ac.jp   aotani@is.titech.ac.jp

Abstract

We propose an advice mechanism called *Chop&Graft* for non-local flow control. It offers a novel `chop` pointcut that lets a piece of advice terminate the current execution, and `graft` and `retry` operators that resume and restart the terminated executions. By using pointcuts for specifying the region of termination, the mechanism is more robust and more concise than the traditional exception handling mechanisms that rely on names or exception classes. The paper presents the design of the mechanism along with the sketches of two implementations using delimited continuations or threads and exceptions.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms Languages, Design

Keywords Aspect-oriented programming, exception handling, delimited continuations

1. Introduction

Non-local flow control is a transition of program execution from a point to another lexically isolated point. Examples are the termination of a task and backtracking. While modern programming languages offer the mechanisms for such flow control, including the exception handlers and continuations, the programs with non-local flow control are often complicated (Lee and Anderson 1990).

Aspect-oriented programming (AOP) enables implementations of non-local flow control to be separated from the core program logic. Previous studies have shown that aspects can implement exception handlers for real-world applications (Lippert and Lopes 2000; Colyer and Clement 2004; Filho et al. 2006; Taveira et al. 2009) as well as contract checkers that raise exceptions (Rebêlo et al. 2010).

Those AOP-based implementations cannot, however, directly realize the control flow that the programmer wants to express. Assume we want to restart a task in the middle of its execution. Even with AOP, we have to encode the flow control by defining two pieces of advice with a loop construct, an exception handler, and code to throw an exception. Since the implementation consists of two pieces of code, it has to pass information from the one to the other by explicitly using the fields of an exception object. Since the implementation relies on the exception handling mechanism in the underlying language, it requires very careful programming in order to avoid the problem of accidental capturing.

We propose a novel advice mechanism for AOP languages, called Chop&Graft, that can express non-local flow control in a more direct way. The key idea is to extend pointcuts so that a single piece of advice can directly express transfer of control from two execution points in a program.

In the following sections, we first present several types of non-local flow control and their AOP implementations (Section 2), followed by the problems of those implementations (Section 3). We then present the Chop&Graft mechanism and how it can solve the aforementioned problems (Section 4). We investigated two implementations, namely a prototype implementation using delimited continuations on top of AspectScheme, and an implementation strategy using threads and exceptions on top of AspectJ with limited capabilities (Section 5). After discussing related work (Section 6), we conclude the paper (Section 7).

2. Traditional Aspects for Non-Local Flow Control

We show four types of non-local flow control, namely *simple task termination*, *task termination with recovery*, *retry*, and *backtracking*. We first give an example program that represents the core program logic, and then illustrate four types of non-local flow control along with implementations in AspectJ where they are possible.

2.1 Core Program Logic: Plugin Loading

Listing 1 is a fragment of a program that initializes plugin modules of an application. Figure 1 is its execution sequence. Since the paper focuses on the control flow, we simplified the example by putting important methods in one class, which is not shown in the listing.

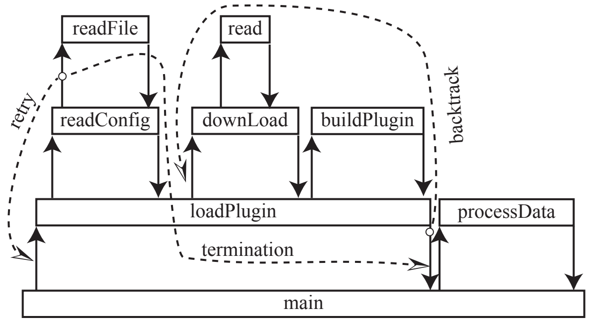

Figure 1. Execution Sequence of the Example Program. (Execution goes from left to right. Rectangles are method executions. Dashed arrows are non-local flow control.)

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-4033-5/16/03...$15.00
http://dx.doi.org/10.1145/2892664.2892674

The program first reads the location of a plugin module from a configuration file. It then downloads the plugin definition (we assume the application can have at most one plugin), constructs a plugin object, and finally executes the body of the application.

2.2 Task Termination

When a task cannot continue in the middle of its execution, a simple resolution is to *terminate* the task and resume the program just after the task. For example, when there is no configuration file during the plugin loading, we terminate the entire plugin initialization task and start the body of the application without plugins.

In AspectJ, we can terminate a task by defining a pair of advice (Listing 2): a piece of advice that wraps the join point that starts the task with a `try-catch` form (*catching advice*), and another piece of advice that `throw`s an exception at the join point where the task can no longer continue (*throwing advice*). This implementation also defines a subclass of `Error` for throwing and catching.

2.3 Task Termination with Recovery

When we terminate a task, we also want to execute a *recovery* task as compensation. For example, when we terminate the plugin loading task, we might want to create a default plugin object.

In AspectJ, the recovery task can be executed in the catching advice (Listing 3). However, if the recovery task needs information at the join point of throwing, it has to pass the information through fields in the exception object, like the `file` field of the `NoConfigError` class in the example.

2.4 Retry

Sometimes, we want to *retry*, i.e., to run the original task (possibly with different parameters) as a recovery task. For example, when a plugin loading task fails, we want to start the task over with a configuration file at a different location (e.g., the home directory).

In AspectJ, we can implement this by wrapping the task execution join point (i.e., `proceed` in the catching advice) with a `try-catch` form in a loop (Listing 4). When the task succeeds, it merely returns from the advice by exiting the loop. When the task fails, the `catch` clause updates the parameters for the next try.

2.5 Backtracking

Backtracking is non-local flow control in an opposite way, where we want to start a task over from the middle of its execution. For example, when a constructed plugin object turns out to be incompatible with the execution platform, we want to restart the task from the join point that downloads the plugin definition by appending a different suffix to the location.

Listing 5 is a core logic rewritten for this flow control. Since it requires to rewind the program control to a point inside of a method call after returning from the method call, AOP implementation is not possible without a special mechanism in the underlying language. The underlined parts are additional to the original application code. Note that the added behavior is to revert to the construction of `InputStream`, after executing `buildPlugin`.

```
void main() {
  Plugin d = loadPlugin(Directory.getCwd());
  bodyOfApplication(d);
}

Plugin loadPlugin(File configDir) {
  URL loc = readConfig(configDir);
  byte[] rawData = download(loc);
  return buildPlugin(rawData);
}

URL readConfig(File dir) {
  File f = new File(dir,"plugin.cnf");
  return new URL(readFile(f));
}

byte[] download(URL url) {
  InputStream s = new InputStream(url.append(SUFFIX));
  return s.read();
}
```

Listing 1. Initialization of a Plugin Module.

```
aspect TerminateLoading {
  static class NoConfigError extends Error {}

  //catching advice
  Plugin around(): call(* *.loadPlugin(File)) {
    try { return proceed(); }
    catch (NoConfigError e) { return null; }
  }

  //throwing advice
  before(File f) : call(* *.readFile(File)) && args(f)
              && if(!f.exists()) {
    throw new NoConfigError();
} }
```

Listing 2. Task Termination Aspect

```
aspect UseDefaultPlugin {
  static class NoConfigError extends Error {
    File file; //initializing constructor omitted
  }

  Plugin around(): call(* *.loadPlugin(File)) {
    try {
      return proceed();
    } catch (NoConfigError e) {
      File f = e.file; //receive information
      return new DefaultPlugin(f.getName()); //recovery
  } }

  before(File f) : call(* *.readFile(File)) && args(f)
              && if(!f.exists()) {
    throw new NoConfigError(f); //pass information
} }
```

Listing 3. Task Termination with Recovery Aspect

3. Problems of the Existing AOP Mechanisms

The implementations of non-local concerns in the previous section are indirect as they express flow control by means of the excepting handling mechanism, and have the following problems.

3.1 Two Pieces for One Control

As we can see in the previous section, each implementation of non-local control with the existing AOP mechanisms is separated into two pieces, namely throwing and catching advice. This is because the implementations rely on the exception mechanism to change the control flow of a program.

This separation makes the program more complicated and hard to understand. It becomes more problematic when the recovery task performed at the catching point requires information from the point of throwing. For example, the aspect implementations in Sections 2.3 and 2.4 had to define a field in the exception class for passing information from the throwing point to the catching point.

```
aspect Retry {
  static class NoConfigError extends Error {}

  Plugin around(File dir)
    : call(* *.loadPlugin(File)) && args(dir) {
    while (true) { //retry loop
      try { return proceed(dir); }
      catch (NoConfigError e) {
        dir = nextSearchPath(dir); //for the next trial
  } } }

  before(File f) : call(* *.readFile(File)) && args(f)
              && if(!f.exists()) {
    throw new NoConfigError();
} }
```

Listing 4. Retry Aspect (For simplicity, we omit the case when no configuration file is found in any search path.)

```
Plugin loadPlugin(File dir) {
  URL loc = readConfig(dir);
  for (String s: SUFFIXES) { //insert a loop
    byte[] rawData = download(loc, s); //add parameter
    Plugin p = buildPlugin(rawData);
    if (Platform.isCompatible(p)) //check the result
      return p;
  }
  return null;
}

private byte[] download(URL url, String suf) {
  InputStream s = new InputStream(url.append(suf));
  return s.read();
}
```

Listing 5. Core Logic Rewritten for Backtracking (changes are underlined)

3.2 Accidental Capturing

Though we could define reusable implementations of non-local control concerns, a solid implementation would become further complicated in order to avoid the *accidental capturing* problem.

First, let us look at a naive reusable aspect with two abstract pointcuts and one abstract method in Listing 6, and a concrete aspect in Listing 7, which implements the same behavior as the one performed by the aspect in Section 2.3.

This implementation is not correct when there is another concrete aspect of GenericTermination that has an overlapping control-flow between the throwing and catching points. This is because they use the same exception class, namely TaskExit, for all concrete aspects. As a result, an exception thrown by CreateDefaultPlugin might be caught by another concrete aspect, or vice versa. We call this the *accidental capturing* problem.[1]

One of the robust implementations would store the concrete aspect instance in an exception object at the throwing point and rethrow the exception if the caught exception object is not thrown by the catching aspect. Concrete aspects can be stored and compared by obtaining a class object through a reflection method, getClass().

4. Chop and Graft

We propose an extended advice mechanism called *Chop&Graft,* which consists of the chop pointcut and the graft operator.

[1] The accidental capturing problem can arise in any program that uses one exception class for more than one overlapping execution points. Our emphasis here is that, an AOP implementation can make the problem more difficult to predict by hiding the use of exceptions from aspect users.

```
abstract aspect GenericTermination {
  abstract pointcut terminate(Object data);
  abstract pointcut taskBegin(); abstract
  Object recover(Object data);

  class TaskExit extends Error {
    Object data; //constructor omitted
  }

  Object around(): taskBegin() {
    try { return proceed(); }
    catch(TaskExit e) { return recover(e.data); }
  }

  before(Object data): terminate(data) {
    throw new TaskExit(data);
} }
```

Listing 6. Generic (yet Dangerous) Termination Aspect

```
aspect CreateDefaultPlugin extends GenericTermination {
  pointcut terminate(Object f): call(* *.readFile(File))
                && args(f) && if(!((File)f).exists());

  pointcut taskBegin(): call(* *.loadPlugin(File));

  Object recover(Object data) {
    File f = (File)data;
    return new DefaultPlugin(f.getName());
} }
```

Listing 7. A Use of the Generic Termination Aspect

4.1 The chop Pointcut

The chop(p) pointcut serves as a modifier to an advice declaration that terminates the current computation up to the join point in the call stack matching p. It takes a sub-pointcut p as an argument, and matches the current join point when there is a join point in the call stack that matches p (similar to cflowbelow(p)). Before the advice runs, the current computation is terminated up to the join point matching p. When the execution of the advice is finished, the computation resumes from the join point matching p.

With a chop pointcut, the termination aspect in Section 2.3 can be rewritten as follows.

```
Plugin around(File f): chop(call(* *.loadPlugin(File)))
       && call(* *.readFile(File)) && args(f)
       && if(!f.exists()) {
  return null;
}
```

When readConfig calls readFile, the advice terminates the execution of readConfig and loadPlugin, and runs the body of the advice, which merely returns null to the caller of loadPlugin.

Compared to the initial aspect definition, a chop pointcut makes it possible to define the termination with one piece of advice.

Task termination with recovery is realized in a straightforward way by executing the recovery task in the advice body:

```
Plugin around(File f): chop(call(* *.loadPlugin(File)))
       && call(* *.readFile(File)) && args(f)
       && if(!f.exists()) {
  return new DefaultPlugin(f.getName());
}
```

With this implementation, the information at terminating (i.e., call to readFile) and at exiting (i.e., call to loadPlugin) can be accessed through advice arguments.

Unlike other kinds of pointcuts, chop is side-effecting. We nevertheless designed it as a pointcut because it should be combined with other pointcuts and it should be able to take information out from the join point matching its sub-pointcut.

When there is more than one join point matching chop pointcuts, the *closest* join point (i.e., the topmost on the call stack) is selected for task termination. This behavior is consistent with the cflow pointcut, which matches the closest join point when it is taking out information from the matching join point.

4.2 The graft Operator

The graft(v) operator can be used in the body of advice with chop and runs the computation terminated by the chop pointcut. The parameter v to graft is used as a return value to the current join point. When the execution of graft comes back to a join point matching the chop pointcut with a return value v', graft(v) returns v' to the caller.

For example, when we want to conditionally terminate the plugin construction task, we write an if statement with a branch that runs the graft operator (which does not terminate the plugin construction) and that just returns with null (which terminates the construction).

```
Plugin around(File f): chop(call(* *.loadPlugin(File)))
       && call(* *.readFile(File)) && args(f)
       && if(!f.exists()) {
  if (DefaultConfig.available()) {
    Plugin p = graft(DefaultConfig.readData());
    p.setDescription("default");
    return p;
  } else return null;
}
```

Since the graft operator behaves just like a method call, advice bodies can perform computation after the execution of graft.

4.3 The retry Operator

The retry operator can only be used in the body of chopping advice and runs the computation at the chopped join point *from the beginning*. For example, in the body of advice with pointcut chop(call(* *.loadPlugin(File))), a retry operator calls the loadPlugin with the same arguments.

The retry aspect in Section 2.4 can be rewritten with the retry operator as follows.

```
Plugin around(File dir, File f):
      chop(call(* *.loadPlugin(File)) && args(dir))
      && call(* *.readFile(File)) && args(f)
      && if(!f.exists()) {
  return retry(nextSearchPath(dir), f);
}
```

It is also possible to explicitly call loadPlugin in the advice body. However, retry makes advice more generic for more complicated cases such as when chop selects more than one join point or when we use an abstract pointcut for chop.

4.4 Discussion: Backtracking and Accidental Capturing

With Chop&Graft, we can implement the behavior by running the graft operator inside a loop.

```
Plugin around(String _s): chop(call(* *.loadPlugin(File)))
       && call(* URL.append(String)) && args(_s) {
  for (String s: SUFFIXES) {
    Plugin p = graft(proceed(s));
    if (p.isValid()) return p;
  }
  return null;
}
```

Since the Chop&Graft mechanism does rely on "names" for transitioning controls, it is safe from accidental capturing. This contrasts with the throw-catch mechanism, which relies on exception classes to distinguish different throw-catch pairs as we pointed out in the previous section.

5. Implementations

We present two implementation approaches: one is to compile with delimited continuations, and the other one is to compile with exceptions and threads. The first one supports all the features of the Chop&Graft mechanism, yet requires delimited continuations in the underlying language. We built an implementation on top of AspectScheme (Dutchyn et al. 2006). The second one does not allow the running of a graft operator more than once, i.e., it does not allow backtracking. Though limited, it merely requires the underlying language to have the standard throw-catch and multi-threading mechanisms. We show this approach by hand-compiling aspects with Chop&Graft into pure AspectJ programs. The implementation of the first approach and the hand-compiled code of the second approach are available at https://bitbucket.org/nikeeshi/chop-graft.

5.1 Compiling with Delimited Continuations

Delimited continuations (Felleisen 1988; Danvy and Filinski 1990) are a functional abstraction of non-local control. They consist of the *shift* and *reset* operators. The shift operator terminates the computation up to the corresponding reset operator and provides the terminated computation as a function. Many implementations of those operators are *tagged* so as to match shift and reset operators without interference.

Our implementation with delimited continuations uses the shift and reset operators for chopping and for performing the graft operator, respectively. We illustrate our implementation strategy with an example in an AspectJ-like hypothetical language with the shift and reset operator.

Assume we have the following advice declaration.

```
T around(...): chop(P1) && P2 {
  ...graft(...)...
}
```

Then the compiler generates a pair of advice declarations. The first one captures join points matching P1 and wraps the join points with the reset operator, where TAG is a unique tag generated for the original advice.

```
T around(...): P1 {
  return reset(TAG, proceed());
}
```

The second advice captures the join point at P2. An additional condition cflowbelow(P1) guarantees that the current join point is surrounded by the reset operator. The body of the advice performs the shift operator and binds the obtained continuation to graft. (Note that (v) -> {...} is a syntax for lambda-expressions.)

```
T around(...): cflowbelow(P1) && P2 {
  shift(TAG, (graft) -> {
    ...graft(...)...
  });
}
```

For languages without delimited continuations, this approach would still be possible by transforming a program to save and restore the current call stack (Sekiguchi et al. 1999) or transforming to a continuation-passing style (Rompf et al. 2009).

5.2 Compiling with Threads and Exceptions

The second implementation compiles Chop&Graft into an AOP language with exceptions and threads, yet without using delimited continuations. The basic idea is to execute the advice body in a newly created thread and let the thread synchronize with the main thread at the beginning and the end of the graft execution.

Listings 8 and 9 outline the compiled advice declarations at the chopping join point and at the join point where the original advice runs.

```
T1 around() : P1() { //at chopping point
  SQ.push();
  try {
    ret = proceed();
    while (!SQ.isEmpty()) {
      <adv,main> = SQ.poll();
      adv.send(GRAFT_END,ret);
      switch (main.take()) {
      case ADVICE_TERM: return ret;
    } }
  } catch (NoGraft ng) {
    return ng.value();
  } finally {
    while (!SQ.isEmpty()) {
      <adv,main> = SQ.poll();
      adv.send(EXCEPTION);
      main.take();
    }
    SQ.pop();
} }
```

Listing 8. Compiled Advice at Chopping Point

```
T2 around(...) : cflowbelow(P1()) && P2(...){
  adv = new Buf();
  main = new Buf();
  SQ.add(adv,main);
  new Thread() {
    void run() {
      try { main.send(ADVICE_TERM,body()); }
      catch (Stop e) { main.send(ADVICE_TERM,null); }
    }
    T2 body() { ...graft(...)... }
    T1 graft(v) {
      main.send(GRAFT,v);
      switch(adv.take()) {
      case GRAFT_END: return adv.value();
      case EXCEPTION: throw new Stop();
  } } }.start();
  switch(main.take()) {
  case GRAFT: return main.value();
  case ADVICE_TERM: throw new NoGraft(main.value());
} }
```

Listing 9. Compiled Advice at Original Advice Point

5.2.1 Stack of Queues for Passing Channels

For each advice declaration with a chop pointcut, we define a global variable for storing a stack of queues. In the compiled code, the variable is represented by SQ. The stack structure keeps track of the topmost join point matching the chop pointcut. (Note that there can be more than one join point that matches the chop pointcut.) The push() and pop() methods of SQ respectively put and remove a queue on top of the stack.

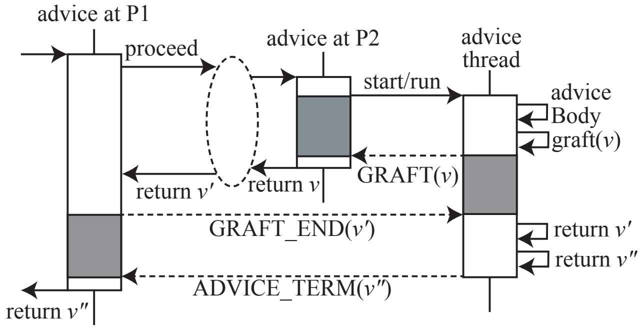

Figure 2. Execution Sequence When Graft is Called (Solid arrows are method calls and returns. Dashed arrows are message sends. Threads are suspended during the gray rectangles. The dashed oval represents intermediate method calls between P1 and P2.)

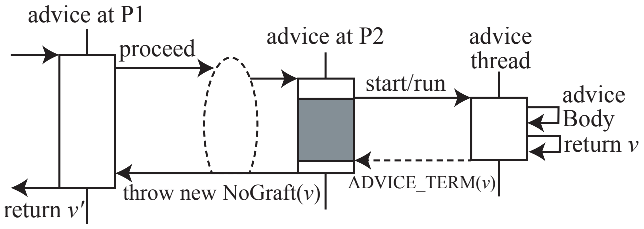

Figure 3. Execution Sequence When Graft is not Called

The top of SQ is used for sending communication buffers from advice join points to the topmost chopping join point. Since more than one advice join point can run before the control comes back to the chopping join point, each element in SQ is a queue. The add, poll, and isEmpty methods of SQ manipulate the queue on top of *SQ*, which respectively adds/removes a pair of buffers to/from the queue and checks the emptiness of the queue. We use a shorthand <adv,main> = SQ.poll() for taking a pair out of the queue and binding it to variables adv and main.

For synchronizing the execution of the advice body (which is running in a separate thread) with the chopping join point, we use a buffer, namely adv in the code. We use shorthands adv.send(LABEL,val) to send a value with a label. adv.take() waits for a message from the buffer and returns the received label. adv.value() returns the received value.

For sending information from the advice execution to the chopping join point, we also use a buffer, namely chop in the compiled code.

5.2.2 Execution with/without Graft

The compiled code embodies two types of behaviors, depending on whether graft is called inside of the advice body or not. The outlines of those behaviors are illustrated as the interaction diagrams in Figures 2 and 3.

When graft(v) is called (Figure 2), the advice thread sends v to the advice join point and suspends its execution. Then the advice returns to the caller (i.e., the chopped operation) with the return value v. When the control reaches the chopping join point with a return value v', it sends v' to the advice thread so that the advice thread resumes with v' as a return value from graft. When the advice thread finishes its execution with a value v'', it sends v'' back the chopped join point.

When the advice thread finishes the execution of the advice body with a return value v, yet without calling graft (Figure 3), it sends v to the main thread. The main thread then throws an exception with v towards the chopping join point. The chopping join point returns v to its caller.

5.2.3 Further Details

The actual compile code handles other corner cases like when graft is called more than once (which will produce a runtime error) and when an exception is raised during the graft operation (which needs to terminate the advice thread).

There are cases where obvious optimizations can be easily implemented. For example, when all graft operations in an advice body appear at tail positions (i.e., they only appear as return graft(...);), we do not need to spawn an advice thread.

6. Related Work

EJFlow (Cacho et al. 2008) enables defining exception handlers as pieces of advice. It offers a concept of exception channels so that handlers can be specified for each pair of exception throwing and catching points. While EJFlow also focuses on the pair of two control points, it still assumes to throw exceptions explicitly. Hence EJFlow does not support flow control beyond termination and recovery and still suffers from an accidental capturing problem.

Loop join points (Harbulot and Gurd 2006), closure join points (Bodden 2011), and region pointcuts (Akai et al. 2009) all extend AOP mechanisms for flow control. These extensions provide abstraction of an execution of a specific region of code so that advice can change additional control flow, such as parallelization. However, those extensions merely handle *local* flow control, i.e., executions of code inside a method.

In AspectJ and its extensions, there are advanced pointcuts, including `cflow`, tracematch (Bodden et al. 2008), and `dflow` (Masuhara and Kawauchi 2003). Those pointcuts resemble Chop&Graft in the ability to express conditions on more than one join point. However, those pointcuts merely serve as predicates, and have no effect on the existing computation.

Delimited continuations are a powerful and flexible abstraction of non-local control. In fact, one of our implementations uses delimited continuations. However, delimited continuations are similar to exception handling mechanisms in that they require writing code at points of throwing and catching and also in that they rely on unique identifiers (called tags) to avoid accidental capturing. Put differently, Chop&Graft can be considered as an AOP-style abstraction of the delimited continuations.

7. Conclusion

We proposed an extended advice mechanism called Chop&Graft for aspect-oriented programming languages. The mechanism consists of the `chop` pointcut and the `graft` and `retry` operators. The `chop` pointcut terminates computation up to the execution point specified by a sub-pointcut, which enables more robust and modularized descriptions of flow control. This is different from traditional mechanisms with exceptions and delimited continuations with or without AOP, where one control concern should be described as two separate descriptions that are paired by using identifiers. Together with the `graft` and `retry` operators, a piece of advice with `chop` can express several kinds of control concerns, such as terminating, retrying, and backtracking.

We implemented the proposed advice extensions on top of AspectScheme with delimited continuations. We also discussed another implementation that only requires threads and exceptions by showing hand-written compiled code in AspectJ. Though we have not yet implemented a compiler of the latter, it gives an essential insight into the implementation.

There are topics left for future work. The first one is to build a concrete compiler implementation by extending an AspectJ compiler and to rewrite existing programs with Chop&Graft. The second one is to investigate the semantics in detail. As we have seen, the compilation into threads and exceptions is fairly complicated. It is not clear what kind of semantic properties hold (e.g., the behavior of a program where the advice with `chop` runs at the body of the advice.) We believe that semantic frameworks developed for delimited continuations would be useful in discussing those kinds of properties. The third one is to study interaction with existing mechanisms for non-local flow control. Since Chop&Graft can be useful only when the programmer knows the both ends of a transition of control, we would use existing mechanisms like exception handlers in other situations. It is an open question, whether a program will behave "without surprises" when Chop&Graft and other non-local flow control mechanisms are used together. Both detailed investigation of formal semantics and user studies will be needed.

Acknowledgments

We would like to thank Izumi Asakura, Matthias Springer, Fernando Castor, and the anonymous reviewers for their constructive comments on the paper.

References

S. Akai, S. Chiba, and M. Nishizawa. Region pointcut for AspectJ. In *Proceedings of Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'09)*, pages 43–48. 2009.

E. Bodden. Closure joinpoints: block joinpoints without surprises. In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD'11)*, pages 117–128, 2011.

E. Bodden, R. Shaikh, and L. Hendren. Relational aspects as tracematches. In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 84–95, 2008.

N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. EJFlow: taming exceptional control flows in aspect-oriented programming. In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 72–83, 2008.

A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 56–65, 2004.

O. Danvy and A. Filinski. Abstracting control. In *Proceedings of Conference on LISP and Functional Programming (LFP'90)*, pages 151–160, 1990.

C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, 2006.

M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of Symposium on Principles of Programming Languages (POPL'88)*, pages 180–190, 1988.

F. C. Filho, et al. Exceptions and aspects: the devil is in the details. In *Proceedings of International Symposium on Foundations of Software Engineering (FSE-14)*,pages 152–156, 2006.

B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 63–74, 2006.

P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 1990.

M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of International Conference on Software Engineering (ICSE'00)*, pages 418–427, 2000.

H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of Asian Symposium on Programming Languages and Systems (APLAS'03)*, pages 105–121, 2003.

H. M. Rebêlo, et al. The contract enforcement aspect pattern. In *Proceedings of Latin American Conference on Pattern Languages of Programs*, article no. 6, 2010.

T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of International Conference on Functional Programming (ICFP '09)*, pages 317–328, 2009.

T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *Proceedings of International Conference on Coordination Models and Languages (COORDINATION'99)*, pages 211–226, 1999.

J. C. Taveira, et al. Assessing intra-application exception handling reuse with aspects. In *Proceedings of Brazilian Symposium on Software Engineering*, pages 22–31, 2009.