An Implementation Mechanism for Tailorable Exceptional Flow

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

***Abstract*—Decomposing software according to the structure of the problem domain makes it easier to manage its complexity. Such a decomposition requires a programming language that supports compositions matching those in the problem domain. However, current programming languages only offer a restricted set of control-flow related composition mechanisms, such as method invocation and exception handling. This limits developers in choosing the desired decomposition.**

Previously, we showed that continuations are suitable for implementing various control-flow compositions. However, composing and refining these implementations requires new language features. In this paper, we outline requirements for control-flow composition and show how exception handling can be implemented using continuations, scopes and bindings.

***Keywords*-language design; continuation-passing style; exception handling; control-flow composition; free composition**

I. Introduction

Software composition is a key element in software engineering and programming language technology: It is the primary means for managing the complexity of large software construction projects [1]. A *composition mechanism* allows programmers to define—at least part of—the behavior of an abstraction in terms of the behavior of one or more other abstractions. Previously, both we [2], [3] and others [4], [5] have argued that restricting the expressiveness of composition offered by a language severely limits developers. Such restrictions require workarounds leading to code duplication and obfuscation of the intention of the composition. Our proposed approach is to allow tailorable composition mechanisms to be implemented as first-class functions. Following such an approach, developers can derive new composition mechanisms if their domain-specific decomposition requires a specific variant. Also, composition mechanisms can be distributed as libraries, allowing developers to choose from them freely.

Previously, we explored expressing various control-flow related semantics of compositions, such as exception handling, method invocations, and data access [6]. We concluded that first-class continuations are suitable for implementing variants of control-flow composition, but new language features are required to compose and refine implementations of such mechanisms so as to avoid redundancy. In this paper, we focus on the ingredients needed for defining and composing such control-flow mechanisms. We outline the requirements and show an envisioned implementation of such mechanisms.

II. Tailorable Control Flow

Our main goal is to allow tailorable control-flow mechanisms inside a single programming language. An example is using multiple kinds of exception handling, such as conventional try-finally and try-with-resources (introduced in Java 7), within the same language. Application programmers should be able to choose which kind of exception handling they want to use. Of course, this choice should be expressed in a programmer-friendly way. Thus, application programmers should be able to think only about the use of exceptions, without dealing with the implementation details. Such implementation details can be left to library implementers. To allow this separation between implementing and using control-flow mechanisms, these mechanisms should be implemented modularly. We propose to implement all control-flow mechanisms as first-class abstractions.

III. Requirements

We identified at least two requirements for implementing the semantics of control-flow mechanisms as first-class abstractions:

- being able to influence the execution order of statements;
- having a way to bind different scopes to blocks so as to identify which variables can be used in which context.

We explain both requirements in more detail in the following two subsections.

A. Influencing Execution Order

Most important for tailorable control flow is the ability to influence the execution order of statements. Listing 1 shows a try-finally statement in pseudo code, which we will use as an example. (Leaving out the catch block is a simplification for the purpose of our discussion.) The statement on line 3 raises an exception, which means that the next statement to be executed is not the sequentially following one (line 4), but the first one of the finally block (line 6).

To make the behavior of the try-finally statement tailorable, it should be implemented as a first-class abstraction.

```
1  var out
2  try {
3    out = File.open("Some file that does not exist") // Raises an exception
4    out.write("Everything went fine")
5  } finally {
6    if (out != null)
7      out.close()
8  }
```

Listing 1. Traditional try-finally

```
1  var out
2  tryFinally({
3    out = File.open("Some file that does not exist") // Raises an exception
4    out.write("Everything went fine")
5  }, {
6    if (out != null)
7      out.close()
8  })
```

Listing 2. First-class tryFinally

We achieve this by implementing it as a function to which both the try and finally block are passed as closures. This first-class implementation of try-finally can be used as shown in listing 2, which is analogous to listing 1. The try-finally statement and its semantics are no longer part of the language definition. Thus, the alternate execution order followed when an exception is raised can be expressed in the language itself; special language constructs are no longer needed. In this example, the semantics of the try-finally statement should be expressed by the implementation of the function tryFinally.

B. Binding Scopes

In the previous subsection, we saw that a try-finally statement contains two blocks: the try and finally blocks. Inside both blocks, it is possible to use variables defined before the try-finally statement, but variables defined inside the try block are not visible by the finally block and vice versa.

However, when we take a look at the try-with-resources statement of Java 7 [7, chapter 14.20.3], scoping is defined differently. A try-with-resources statement allows a set of AutoCloseable resources to be given, which are closed automatically when exiting the try block. Listing 3 shows an example of try-with-resources, which has the same behavior as the example described in the previous subsection: a file is opened (line 2), something is written to that file if opening succeeds (line 4), and finally the file is closed (implicitly when execution of the try-with-resources statement finishes).

For this try-with-resources statement, we can define a first-class implementation, which can be used as shown in listing 4. Here, line 2 represents the resources block and line 4 the try block. We see that the try block can not only use the variables defined before the try-with-resources statement, but also the variables defined within the resources block.

The try-with-resources scoping differs from that of try-finally, but scoping is not limited to these two variants. For example, if a finally or catch block is added to the try-with-resources statement, variables defined in the resources block could be either accessible or inaccessible from the newly added block. Thus, we see that multiple scoping choices exist, which should not be limited to a fixed set provided by the language. Therefore, we conclude that *scoping should be tailorable*.

When scoping is tailorable, library implementers should also have the means to express which block uses what scope. Therefore, *library implementers should be able to bind scopes explicitly*.

IV. Solution approach

A. Influencing Execution Order

In earlier work [6], we showed that the continuation-passing style is a powerful mechanism for influencing execution order. However, programming in this style is—in general—less readable than programming in a direct style. Therefore, application programmers should not have to program in the full continuation-passing style. However, library implementers should be able to influence the execution order. For example, for the try-finally statement, library implementers can define the concept throw as a continuation. Once defined, such a continuation is available implicitly to application programmers. Thus, when using the try-finally control-flow mechanism, application programmers can invoke throw() to raise an exception, without writing all code in the continuation passing style.

B. Scopes

Most programming languages structure scopes as a tree or forest, in which every scope has at most a single enclosing scope. In general, these scopes are lexically nested. Listing 5 shows an example of nesting; the boxes depict the scopes and the arrows the nesting (they point towards the enclosing scope). Such a tree structure is sufficiently powerful for modeling the scopes of a wide range of control-flow mechanisms. However, as shown in section III-B and in listing 6, depending on the control-flow mechanism, the choice of enclosing scope might vary. Therefore, we propose using first-class scopes. These scopes become accessible properties of continuations, of which library implementers can alter the enclosing scope. In this way, scopes can be organized in any desired tree structure.

Theoretically, it is possible to create a cyclic graph if enclosing scopes can be freely reassigned. However, cycles can lead to infinite recursion during variable lookup, when referring to an undefined variable. Such infinite recursion can be avoided, but since we have not seen any use cases of

```
1  try (
2    var out = File.open(...)
3  ) {
4    out.write("Everything went fine")
5  }
```

Listing 3. Java-like try-with-resources

```
1  tryWithResources({
2    var out = File.open(...)
3  }, {
4    out.write("Everything went fine")
5  })
```

Listing 4. First-class try-with-resources

```
1  tryWithResources(
2    { var out = File.open(...) }
3    ,
4    { out.write("Everything went fine") }
5    )
```

Listing 5. Default scoping

```
1  tryWithResources(
2    { var out = File.open(...) }
3    ,
4    { out.write("Everything went fine") }
5    )
```

Listing 6. Desired scoping

cyclic scopes, implementations might disallow cycles in the scope graph. In that case, the scope graph will be a directed acyclic graph, which—together with the requirement that every scope has at most one enclosing scope—implies that the scope graph is a forest.

C. Bindings

The previous two subsections explained the purpose of both continuations and scopes when *using* a control-flow mechanism. Now, we will show an example of their purpose while *defining* a control-flow mechanism. This example is the try-finally statement, which was explained in section III-A.

Listing 7 shows an envisioned implementation of tryFinally. Lines 2–11 specify how scopes and continuations are bound, termed *bindings*, and lines 13–14 specify the actual function body.

Bindings can refer to continuations that are either passed as arguments (try and finally, in this example) or available implicitly (return and throw, in this example). Bindings can both call these continuations and access their properties, which are either scopes or continuations. Common scopes are: the scope created by a continuation (scope), the enclosing scope used during execution of a continuation (enclosingScope), and the scope in which the continuation is defined (definingScope). Common continuations are return and throw, which represent the behavior of returning from a method and raising an exception, respectively.

Looking at the listing, we see that both the try and finally blocks close over the scope in which they are defined (lines 2 and 3). Further, inside the try block, return and throw cause finally to be executed before actually returning or throwing, respectively. For example, lines 4–7 specify the continuation return, which receives the value to be returned (line 4) and executes finally (line 5) before actually returning this value (line 6).

Our proposed implementation of tryWithResources uses different scoping. When we look at line 3 of listing 8, we see that the try block closes over the scope used by the resources block. Further, instead of calling finally, it closes all defined resources (lines 17–18). The precise way of accessing all defined variables is future work.

V. Related Work

We outlined an implementation of tailorable control-flow mechanisms. These mechanisms are implemented as functions that receive closures as arguments, which represent the blocks to execute. The control-flow mechanisms can control the scope in which these closures are executed.

In this section, we discuss programming language features on which we might be able to base our approach. For this purpose, we show various ways of representing control state as first-class values and the differences in binding scopes.

JavaScript [8] provides the function eval, that receives a string as argument and executes its contents as if it is source code. When defining blocks as strings, the execution order of statements can be changed by passing these blocks around.

```
1  function main() {
2      var s = "This value is never used";
3      var block = 'alert(s); s = "This value also will never be used";';
4      s = "This will be printed second";
5      printData(block);
6      alert(s);
7  }
8  function printData(block) {
9      var s = "This will be printed first";
10     eval(block);
11 }
```

Line 3 of the example above defines the variable block and assigns a string to it, which contains JavaScript source code. Line 5 passes this variable to the function printData. Line 10 evaluates the variable block, executing its contents as if it is source code. The first statement in block—alert(s)—prints the contents of variable s. Since the statements in block use the scope where block is evaluated, this prints: “This will be printed first”. When the alert statement on line 6 executes, s has the value “This will be printed second”, since variable s in the method main will not be affected by the assignment in the printData function.

C# [9] provides delegates, which are type safe pointers to a block of code and can be passed around. Delegates close over the scope in which they are defined. Therefore, it is

```
method tryFinally(var try, var finally) [
    try.enclosingScope = try.definingScope
    finally.enclosingScope = finally.definingScope
    try.return = { value →
      finally()
      return(value)
    }
    try.throw = { exception →
      finally()
      throw(exception)
    }
] {
  try()
  finally()
}
```

Listing 7. Definition of tryFinally

```
method tryWithResources(var resources, var try) [
    resources.enclosingScope = resources.definingScope
    try.enclosingScope = resources.scope
    try.return = { value →
      for (resource defined in resources)
        resource.close()
      return(value)
    }
    try.throw = { exception →
      for (resource defined in resources)
        resource.close()
      throw(exception)
    }
] {
  resources()
  try()
  for (resource defined in resources)
    resource.close()
}
```

Listing 8. Definition of tryWithResources

possible to invoke delegates that use variables that are not directly accessible from the scope that invokes the delegate.

```
public delegate void delegateBlock(); // Declare the signature of the
      delegate
static void Main(string[] args) {
    String s = "This value is never used";
    printDelegate block = delegate { Console.Out.WriteLine(s); s = "This is
          now the new value of s"; };
    s = "This string is printed";
    printData(block);
    Console.Out.WriteLine(s);
}
public static void printData(printDelegate block) {
    String s = "This string isn't printed";
    block();
}
```

Line 4 of the above example creates the delegate block inside the scope of Main. Line 6 passes block to the printData function, which invokes block on line 11. Variable s used in the delegate block refers to the value of s in the scope of Main, where the delegate was created. Therefore, the execution of block will print "This string is printed". Then, block changes the value of s to "This is now the new value of s", which is printed on line 7. The variable s instantiated on line 10 is never used in this program.

Scala [10] offers delimited continuations, which allows limiting the boundary of the continuation-passing style. This means that when a continuation is created, only part of the program is passed as argument to this continuation. Scala provides two functions to achieve this: reset and shift. reset limits the boundary of the continuation-passing style. shift passes the continuation to its body. This continuation represents the rest of the code until the end of the reset block.

```
var s = "This value is never used"
reset {
  s = "This will be printed first"
  shift { (block: Unit => Unit) =>
    println(s)
    block()
    println(s)
  }
  s = "This will be printed second"
}
```

When the code in this example is executed and gets to line 4, shift captures the closure representing the rest of the code in the reset block (only line 9) and passes it as the argument, named block. Then, line 5 prints the variable s that has the value assigned at line 3. Line 6 calls the continuation, resulting in the execution of line 9. Finally, line 7 prints the value of s as assigned on line 9.

VI. Summary and Future Work

In this paper, we presented requirements for a programming language that allows to tailor control-flow mechanisms. We outlined a solution in which control-flow mechanisms are implemented as functions that receive continuations as representations of blocks to execute. We also showed how the envisioned language can control which continuations are passed and in which scope these continuations are executed. Defining the precise semantics of this envisioned language is future work.

References

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

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

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

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

[5] Brooks, F.: No silver bullet: Essence and accidents of software engineering. IEEE computer (1987)

[6] te Brinke, S., Bockisch, C.M., Bergmans, L.M.J.: Reuse of continuation-based control-flow abstractions. In: Proceedings of the 2nd Workshop on Free Composition @ Onward! 2011. FREECO-Onward! '11, New York, NY, USA, ACM (October 2011) 13–18

[7] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java™ Language Specification: Java SE 7 Edition. Oracle (February 2012)

[8] ECMA: Standard ECMA-262 – ECMAScript Language Specification. 5.1 edn. (June 2011)

[9] ECMA: Standard ECMA-334 – C# Language Specification. 4 edn. (June 2006)

[10] Odersky, M.: The Scala Language Specification, Version 2.9. Programming Methods Laboratory (May 2011)