Generating Continuation Passing Style Code for the Co-op Language

Mark Laarakkers

University of Twente
Faculty: Computer Science
Chair: Software engineering

Graduation committee:
dr.ing. C.M. Bockisch
dr.ir. L.M.J. Bergmans
Ir. S. te Brinke

June 12, 2013

Contents

Chapter 1

Introduction

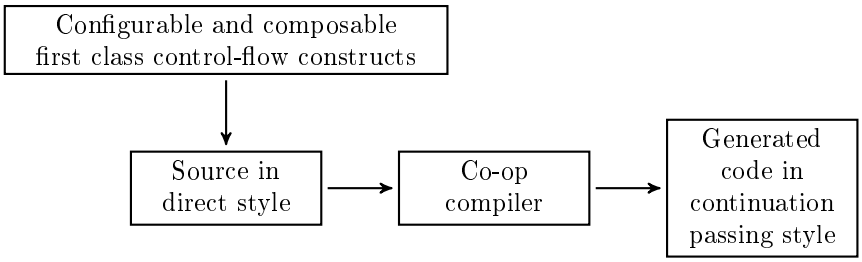
We want to be able to create first class composable and configurable control-flow constructs (See section 1.1.1) in the Co-op language (section 1.1.2). To support this we need first class control state. As shown in a previous research [1], continuation passing style (See section 1.1.4) is a suitable solution to be able to offer first class control flow state.

In the future we would like to have Co-op with minimal built-in control flow constructs and let the programmer create their own constructs. The "standard" control flow constructs such as if statements and loops could be provided as functions in a standard library as opposed to having them built in as keywords in the language. The original Co-op implementation provides no way of creating such control flow constructs. In the generated direct style code there is no way to use control state as first class objects so it won't be possible to let programmers create new control flow constructs without changing the grammar or the compiler. The currently used control flow constructs in Co-op are keywords in the language.

To be able to let programmers that use Co-op create these control flow functions, the generated code by Co-op has to be in a form that allows the use of first class control state. Later this first class control state could be made accessible to programmers so they can use it to create first class control flow constructs.

During the preparation Co-op to be able to create fully first class control flow constructs we will only focus on the generation of the code. We do not want to make changes to the grammar of Co-op and also do not want to change it to a language that forces the use of CPS. The use of CPS in the generated code should be transparent to the programmers so that the programmers will still be able to write in a more natural and easier to read direct style.

1.1 Background information



We want to have support for configurable and composable first class control-flow constructs in the source language of Co-op. The source language is in direct style and conforms to the Co-op grammar. This source language is the input for the Co-op compiler, which outputs generated code. We want this generated code to be in continuation passing style.

1.1.1 Configurable and composable first class control-flow constructs

Configurable and composable first class control flow constructs [2] are ways to be able to pass around the control flow as first class objects. An example of how you could use such a construct is a function that implements the behaviour of an if statement. Such function takes a Boolean value, and an closure argument that represent the code that needs to be executed if the Boolean is true. A closure contains the code to be executed, as well as a referencing environment for the variables used in the closure.

```
1 function if(Boolean b, closure ExecuteIfTrue);
```

The function if in the listing above takes a first class representation of control flow, namely the parameter ExecuteIfTrue. This configurable and composable first class control-flow construct can be used to create these control flow statements. This eliminates the limitation of only being able to use a subset of the possible control flow constructs that are accessible by using a restricted set of keywords.

1.1.2 Co-op

Co-op [3] [4] is a programming language developed for the definition and construction of composition operators. It is a general purpose programming language.

Why we want controllable and composable first class control-flow in Co-op

Co-op is described by the creators as:

> "Co-op is a workbench for the definition and use of composition operators: abstractions that can encapsulate standard solutions such as coding idioms, design patterns and composition techniques, and can later be (re-) used just like library classes."

Decomposing software is used to better match the solution to the problem domain. Decomposition allows the programmer to break down things in to smaller pieces that handle certain behaviour. So Co-op offers a lot of flexibility in this domain. However, Co-op, like other current programming languages, only offers a restricted set of control-flow composition mechanics. This limits the developer in choosing the desired decomposition. An example of such a control flow mechanic that a programmer might want to use is an equivalent of the try-with-resources [5] statement that was introduced in Java 7. try-with-resources is a variant on the conventional try-finally that automatically closes closable resources (that are created as an argument to the try block) after the execution of the try block. It is possible to create this in the Co-op language by extending the language. A keyword will have to be created and the grammar will have to be parsed differently to create the desired behaviour. But we want to be able to create this try-with-resources function, as well as other possible control flow constructs, without having to change the grammar when implementing extra constructs. Listing 1.1 contains an example of calling a function that uses first class control constructs as parameters, then tryFinally is a identifier instead of a keyword.

```
1 var tryblock = { var out = File.Open(...); File.Writeln(...) };
2 var catchblock = { write("something went wrong"); };
3 var finallyBlock = { write("done") };
4
5 tryFinally(tryblock, catchblock, finallyblock);
```

Listing 1.1: The calling of a first class tryFinally function

1.1.3 Steps of compilation

The Co-op compiler of a language takes two steps to create output from input code.

1. The input of the programmer, when being in the correct syntax as defined by the Co-op grammar, is parsed into an abstract syntax tree (AST) containing all elements of the source code as elements that are defined in the grammar.
2. The abstract syntax tree is used as the input for the generator. This generator will create output for the elements in the AST to output, by

traversing the tree. The output in the case of Co-op is Java code in direct style.

1.1.4 Continuation passing style

Continuation passing style (CPS) [6] [7] is a programming style in which the control flow of a program is passed explicitly to function calls. In this style of programming a function will never return. Instead of returning, the program will execute the passed control flow. If the passed control flow is empty, the program is done and terminates. The entire program will look different from a program written in the more conventional direct style. Each function will have an additional argument that is the control flow. When a function would return in direct style (whether implicit at the end of a function or explicit, using a return statement) the continuation will be invoked. Because of this, there can be no code after the execution of a function. Code that will need to be executed after the function call will be passed as the continuation. The example in listing 1.2 is an example program written in pseudo code for a direct style programming language.

```
1  class example {
2
3    int doubleNumber(int n)
4    {
5      return n*2;
6    }
7
8    void main()
9    {
10     int i = 5;
11     i = doubleNumber(i);
12     print(i);
13   }
14 }
```

Listing 1.2: Example program written in direct style

```
1  class example {
2
3    int doubleNumber(int n, closure c)
4    {
5      //this will invoke the closure c with as argument n*2
6              c(n*2);
7    }
8
9    void main()
10   {
11     int i = 5;
12             doubleNumber(i, { print(i) } );
13   }
14 }
```

Listing 1.3: Example of the program in listing 1.2 in CPS

The function doubleNumber on line 3 in listing 1.3 now takes as second argument a continuation. Instead of returning, the second argument of doubleNumber that is called on line 12, the continuation ({ print(i) }), is executed. The difference

is visible when looking at the control flow of the two programs. In direct style the control flow goes from the main function to the doubleNumber function. When this function returns the control flows goes back to the main function. After executing the print function the control flow, again, returns to the main function.

```
1 main
2   doubleNumber
3 main
4   print
5 main
```

Listing 1.4: The control flow of the example program in direct style

The control flow of the CPS also starts in the main function, and then goes to the doubleNumber function. Instead of returning to the main function when doubleNumber is done it invokes the continuation of the rest of the program. This will cause the control flow to go to the print function. After the print function is done the continuation of the rest of the program is empty and the program will terminate.

```
1 main
2   doubleNumber
3     print
```

Listing 1.5: The control flow of the example program in continuation passing style

1.2 Problem statement

We categorize programmers with respect to using the Co-op language into three categories.

- Co-op language developers. People who develop the Co-op language, they have the power to change the Co-op language.
- Expert Co-op users. These users use the Co-op language, they can change the Co-op language (as it is open source). They can also create new constructs and mechanics using the Co-op language. For example new composition mechanics or the control flow functions.
- Co-op users. Programmers using the Co-op language. They only use the language and will not change the source code of the Co-op language. They also will only use constructs and composition operators defined by expert users.

Currently, to implement a first class TryFinally function the programmer (Expert Co-op user, or Co-op language developer) would have to change the grammar and compiler of Co-op to specifically add the TryFinally function. We want to allow expert Co-op users to be able to create functions such as the TryFinally function without having to change the grammar or compiler of Co-op or affecting Co-op users.

The Co-op compiler, offers no support for first class control flow constructs. To prepare the Co-op language we are going to change the compiler to output a format that has support for first class control flow constructs. The format we have chosen to output is continuation passing style. CPS offers us support for first class control flow constructs.

We must choose a language and technique we are going to use to create the CPS version of Co-op. Creating the CPS version of Co-op will not be trivial, we have to think of a CPS solution for every construct used in the Co-op language.

Another requirement we have is that we want full backwards compatibility with the original Co-op language. We are not going to change the syntax so we still support the original syntax. In the original version of Co-op however, we are able to call Java library functions. The CPS version of Co-op should still be able to call the same functions, even though the CPS version might not be in Java.

1.3 Research questions

What do we need to do to make the Co-op compiler generate code in continuation passing style? To answer this question we first have to answer these sub questions.

1.3.1 Q1. What target language and/or technique to use?

Co-op generates to direct style Java code. What will be a suitable target to generate CPS style code; What languages might be suitable that offer multiple techniques to better make this transition?

1.3.2 Q2. How can we use API and system functions, that are not in CPS, in the CPS version of Co-op?

Co-op uses functions that are in the standard Java API, or third party libraries for some functionality. An example is the use of the standard collection classes of Java. We would like to be able to use these collection classes in our Co-op code. The functions in these collections are not in continuation passing style, they do not accept a continuation parameter but execute code and then return. How can we use and call these functions in the CPS version of Co-op while staying as close to pure continuation passing style as possible?

1.3.3 Q3. How can we keep the CPS version of Co-op compatible with code written for the non CPS version?

Programs written in Co-op before changing to CPS should still compile with the new version of Co-op. If these programs can still all be compiled and have the same behaviour then we will have compatibility with the original Co-op. If we succeed then there won't be noticeable changes for the Co-op users.

1.4 Approach

We are going to answer the research questions by one by one, as described in the following sections. In the chapter Evaluation & Conclusion we are going to evaluate our decisions.

1.4.1 Q1. What target language and/or technique to use?

We are going to look at what languages and techniques supported by that language are viable targets for the Co-op compiler to compile to. We are going to look at some languages identified by a previous study [1] and list their advantages and disadvantages and then choose a language to use.

1.4.2 Q2. How can we use API and system functions, that are not in CPS, in the CPS version of Co-op?

For full compatibility we have to support API and system functions used in Co-op. We do not have control over these functions and therefore cannot change these functions to a CPS version. We are going to look at how we can call these non CPS functions while staying as close to pure CPS code as possible.

1.4.3 Q3. How can we keep the CPS version of Co-op compatible with code written for the non cps version?

We are going to change the code generation part of the Co-op compiler. Co-op has built in test cases to test if the compiler works correctly. We consider this goal of compatibility reached if all the programs in the Co-op test suite compile without errors and have the same behaviour when run.

Chapter 2

What target language and/or technique to use?

2.1 Original Co-op implementation

The original Co-op implementation compiles to Java code in direct style. We want to change the output to something in continuation passing style. In this chapter we are going to investigate what requirements we have for the language and technique to create this new version of Co-op. We will see what will be a suitable output language for this CPS version of Co-op. The languages we are going to investigate can have multiple different techniques we are able to use to create CPS code. We will also have to look at the advantages and disadvantages of using a certain technique.

2.2 Approach

After having chosen a target language and technique we will change the Co-op language in two steps. First, we are going to create a version of Co-op which compiles to the chosen target language, without converting to CPS. We do this because changing the generation to CPS will not be trivial. And we want to be able to do the changes in small increments. To do this we need to have a working version of Co-op that generates the chosen language. When we have this version we can start to change different constructs we generate to be in CPS. Using this technique it will be easier for us to create the CPS version of Co-op. We want this intermidiate version of the Co-op language because we already have a working version of the Co-op compiler, otherwise we would start from scratch.

2.3 Requirements

We have the following requirements that a language and technique needs to support to be a viable candidate for our target language.

- R1. Must be possible to output CPS
  The main goal is to be able to output code in CPS. For a language to be able to support code in CPS we need be able to encapsulate the behaviour of the rest of the program in a construct we can pass around. We can pass this construct to function calls so the function has access to the rest of the program, and because of this the function does not need to return, instead it invokes the construct passed, that represents the continuation. We are going to investigate what techniques we can use to achieve this behaviour.
- R2. Backwards compatibility with the Original Co-op implementation
  We would like to have backwards compatibility for programs written in the original version of Co-op. In the original Co-op implementation we are able to use functions from Java libraries, so we also want to be able to do this in the new version of Co-op.

Aside from these requirements we also look at other things to determine what language and technique we are going to use.

- R3. Support in Eclipse
  As Co-op is written within Eclipse using XText [8], having the target language supported in Eclipse is an advantage as we use one tool for everything.
- R4. Ease of development
  How hard is it to use these techniques, and debug the compiled program, to see if the Co-op generator generates correct code. For this readability of the output of the generator is important, as it is easier to debug the code if it is readable.

2.4 Possible candidates

We investigated three possible candidates. Aside from Java, which we investigate because it is the language used in the original Co-op implementation, we also investigate Scala and C#. In previous research [1] we found that both Scala and C# offer techniques that could be used to generate CPS code. These possible candidates are discussed in the next sections of this chapter.

2.5 Java

The target language of the original Co-op implementation. It would be easier to start on a CPS version using Java as opposed to using another target language, as we do not have to create an intermediate version (see section 2.2).

2.5.1 Technique: Function objects

A technique that Java supports is the usage of Function objects [9], in the form of local classes. This allows classes to be defined inside methods. We can use these classes as closures. If we would create an interface called Continuation that requires the method invoke to be implemented by classes who implement the interface Continuation, we can use the classes that implement Continuation to pass around as continuations of the rest of the program.

```
1  public static void main(String [] args) {
2      int i = 0;
3    Continuation cont = new Continuation() {
4        public void invoke()
5        {
6          //code that is the continuation of the rest of the program
7        }
8    }
9
10   functionCall(cont); //Pass the continuations encapsulated in the cont object to the function
         functionCall
11 }
```

Listing 2.1: An example of using Local classes in Java

A problem with these local classes is that the code inside the local classes can not access variables defined outside the scope of the local class, unless it is a final variable. In the example in listing 2.1, the code in the invoke method of the Continuation object cont on line 6 is not able to access the variable i defined on line 2. To overcome this we would need to generate some code that would pass the relevant variables to the continuation to be able to use them. To do this we would need to add boiler plate code for this. Adding this boiler plate code would make the generation harder, as we would have to keep track of the variables we need to access in the continuation. Also this would make the generated code cluttered and harder to read, and thus to debug.

Advantages

- (R4) As the original Co-op implementation is in Java we would not need to create an intermediate version as we described in section 2.2. This would possibly save some development time.
- (R2) Easy to guarantee backwards compatibility as we can call methods on Java objects.

- (R3) Good integration in Eclipse. Co-op is developed in Eclipse with XText. It is easy to have all steps of the process (parsing, generating, compiling output, and running the program) in one application.

Disadvantages

- (R4) Have to add extra behaviour for using variables in the local classes.
- (R4) Generated code will have to contain boiler plate code for variable access, making it more cluttered and harder to read.

2.6 Scala

Scala [10] is a general purpose language, built on top of the Java virtual machine, maintaining strong interoperability with Java. This enables us to call functions on Java classes directly in Scala, thus making backwards compatibility easy. Scala offers programming constructs and techniques that are not available in Java. We are going to investigate two of these techniques, Closures and continuations.

2.6.1 Technique: (delimited) Continuations

Scala offers delimited continuations [10] [11]. A delimited continuation gives Scala the option of enclosing the boundary for the continuation so that when the continuation is called, only a part of the program is passed as the continuation. There are two functions for this, reset and shift. reset encloses the continuation. shift passes the continuation to its body, the continuation is the closure of the rest of the code until the end of the reset function. As a side note in Scala the curly brackets are used to enclose function arguments.

```
1  var s = "This value is never used"
2  reset {
3      s = "This will be printed first"
4      shift { (block: Unit => Unit) =>
5      println(s)
6      block()
7      println(s)
8      }
9    s = "This will be printed second";
10 }
```

Listing 2.2: An example of using delimited continuations in Scala

As the code in the example shown in listing 2.2 is executed and gets to line 4, the shift function will take the closure of the rest of code in the reset function argument (only line 9) and use it is as an argument. Then line 5 will get executed, printing the variable s with the value assigned at line 3. At line 6 the

continuation is executed, resulting in executing line 9. Then line 7 is executed, printing the value of s as assigned on line 9.

A disadvantage of this approach is that it is hard to read as we would get a lot of nested reset statements, and the part that is the continuation is not clearly marked.

Advantages

- (R2) (R3) Scala has great Java integration

Disadvantages

- (R4) Hard to read code
- Need for an intermediate Co-op version (see section 2.2)

2.6.2 Technique: Closures

Another technique Scala offers are closures [12], Closures are first class objects which represents code. The code in a closure is able to access all variables that are accessible in the scope in which the closure is created.

```
1 def main(args: Array[String]): Unit = {
2     var i = 0;
3     var cont = (Any : Any) => {
4                                 //code that is the continuation of the rest of the program
5                                 }
6   functionCall(cont); //Pass the continuation encapsulated in the cont closure to the function
          functionCall
7   }
```

Listing 2.3: An example of using closures Scala

Unlike Java local classes, the code in the closure cont, defined on line 3 in listing 2.3 can access variables that where accessible in the scope in which the closure was created. The closure created on line 3 in listing 2.3 can access the variable i defined on line 2. Closures are not able to use variables that are accessible in the execution scope of the closure, but were not accessible in the place the closure was created.

Advantages

- (R2) (R3) Scala has great Java integration
- (R4) Closures are easy to create and pass around, and have access to variables that were accessible in the context where the closure was created
- (R4) Better readability for the code than (delemited) continuations

Disadvantages

- Need for an intermediate Co-op version (see section 2.2)

2.7 C#

C# is another popular general purpose programming language. It offers delegates [13] which are closures. A problem might be to keep backwards compatibility as C# does not offer a way to call Java libraries without extra tools.

2.7.1 Technique: Delegates

Delegates [13] give us a first class object which represents code, and the codes referencing environment. The code in a closure is able to access all variables that are accessible in the scope in which the delegate is created. Delegates can not access variables that are accessible in the scope were the delegate is invoked, but where not accessible in the scope where the delegate was created.

```
1  public delegate Object continuationDelegate();
2
3  static void Main(string[] args)
4  {
5    int i = 0;
6      continuationDelegate cont = delegate { //code that is the continuation of the rest of the program
7                                 //has to return a value since continuationDelegate returns a object
8                               return null; };
9    functionCall(cont); //Pass the continuation encapsulated in the cont closure to the function
           functionCall
10 }
```

Listing 2.4: An example of using closures Scala

Like the Scala closure example, the code in the delegate cont, defined on line 6 in listing 2.4 can access variables that where accessible in the scope in which the closure was created. The continuation created on line 6 in listing 2.4 can access the variable i defined on line 5.

Advantages

- (R4) Delegates are easy to use closures that we could use to create a CPS version of Co-op.

Disadvantages

- (R3) No support in eclipse.
- (R3) Having to use multiple tools for development in Co-op.

- (R2) Hard to support backwards compatibility. No native support for calling Java libraries, workarounds are possible, but it will not be as easy as with language that natively supports it.

2.8 Technique chosen

We have decided to use Scala with Closures. Closures will help us to create a CPS version of Co-op without the clutter using Java local classes or Scala continuations would require. Also, using Scala, we have the advantage of easy backwards compatibility, as Scala can call Java libraries natively. There is a Scala plugin for Eclipse, so we have Eclipse integration and therefore only need one tool for development.

One of the inherent problems of using closures to create CPS code is that we never return, and therefore the stack will keep growing, and cause a stack overflow. We have not considered this while choosing the technique to use, but it can become a problem later on.

Chapter 3

Changing the generator

Co-op source code is parsed to an abstract syntax tree (AST), which is the input of the generator

Because the original Co-op compiler compiles to Java source code, the output of the generator can run in the Java VM. We are going to change this generator to, output Scala code (See chapter 2), and instead of direct style, we output CPS. In this chapter we are going to list the changes we need to do to create the correct output. We will specify what the generator needs to do, and in difficult cases, how it needs to do that. We will, however, not look at the code of the generator, but at the concepts it uses.

When generating output for the input AST, we will traverse the AST. We will map the elements in the AST to output code. The generation was straightforward. If the original Co-op implementation has to map a method call to a method named `m`, then we could map this method call to the code `m()`. In our new version of Co-op, that outputs Scala code in CPS this is not as straightforward. Because the new output should be in CPS we can not simply map this to a method call `m()` as we could in the original Co-op implementation. In the generation we have to distinguish between different cases to use different generation strategies. We have to generate different code for the mapping of the function call `m` if the function call is an isolated statement, as opposed to when the function call is the argument to another function call. There exist a lot more of these context dependencies for other elements. These context dependencies for generating will make the generator complex, being context dependent means that we need different generation strategies for the same AST element, based on the context in which the element exists.

In this chapter we are first going to give some background information to help you to understand the generator. Then we are going to investigate different elements for which we have to generate CPS code. We are going to see what difficulties there are when generating the elements, and what possible solutions

are to transform them to CPS, as opposed to the direct style in which they are defined in Co-op.

3.1 Background information for the generator

3.1.1 Naming in the generated code

The Co-op generator renames variables and function calls to ensure we do not get naming conflicts in the generated code. We could get naming conflicts when we have multiple functions with the same name but in different classes. This could lead to having multiple functions with the same signature in CoopObject (see section 4.2). Another possibility of having naming conflicts is when elements in the Co-op code are named with names that are valid in Co-op but are not in the code we generate. For example, if we would name a class void, this is not a reserved keyword in Co-op but is a keyword in the language we generate to, therefore we have to rename the variable. The renamed elements are recognizable by the $ symbol in the name which is a symbol that is not allowed to be used in naming in the Co-op language.

3.1.2 CPS closures

We implement CPS in our Scala version of Co-op by using closures. Closures are a way to have pieces of code first class. Such code can be executable by invoking the closure. A closure closes over the environment when created, that means when creating a closure, the closure can resolve all variables and identifiers that are resolvable at the creation site. The variables used by the closure always reference the variables at the creation site; it is impossible to use a variable that exists in the execution context, when invoking a continuation. The closure only has access to the environment in which it was created, rather than the environment in which it is executed.

The closures are used to represent the continuation of the program. We can create a closure, that contains the code that needs to be executed as the rest of the program. It also contains the reference environment for this code. Since the closure is a first class construct we can pass it around. When we need to execute the rest of the program, we can do this by invoking the continuation.

The closures in Scala have the following form as shown in the listing below.

```
1 (<variablename>: <type>) => { <code of continuation> }
```

The <variablename> is the formal parameter to the continuation. type is the type of this parameter. <code of continuation> contains the code that is executed in the closure.

To give an example how these closures work we have the following code in the following listing.

```
1  def functionA(cont : Any => Any)
2  {
3    var i = 2;
4    cont();
5  }
6
7  def main(args: Array[String]): Unit = {
8    var i = 1;
9      var cont = (variablename : Any) => { println(i); }
10     functionA(cont);
11 }
```

Listing 3.1: An example of using closures in Scala

In the listing above we create a Closure on line 9; The closure contains the code println(i);. The function functionA has the cont argument. This argument is of the type Any => Any. This means the input type of the continuation is Any and so is the output type. We pass the closure created on line 9, to the function functionA. In functionA on line 4 we invoke the closure. This will print the variable i. It will print the variable from the environment it is created in, when the closure is created it binds the variable i to that defined in main on line 8; Therefore, executing this program will print 1, instead of using the variable i with value 2, defined on line 3, as this variable is not bound to the closure.

Every function in our compiled Co-op program will have a parameter to accept a closure. This closure contains the continuation of the rest of the program. So instead of returning at the end of the function, we can invoke the closure of the continuation to have the program be in CPS.

3.1.3 invokeCont

invokeCont is a helper function. This function is used in the code the generator outputs, to make code generation easier by not having to distinguish between the format of the closure. In the generated code we have closures in two different formats. In the first format we have the closure as a variable. In the second format the continuations have the form (Any: Any) => { //continuation }. We can invoke the closures in the first form by calling <variablename>(). This is not possible for the second form. To not have to worry about the form of the continuation we use the invokeCont function, that is defined in the following listing.

```
1  def invokeCont(cont: Any => Any)
2  {
3    cont();
4  }
```

We can pass both forms of closures as the argument to the invokeCont function, and they will be invoked. Therefore, because we use invokeCont, we do not have to distinguish between the two forms of closures we generate.

3.2 Elements to map

3.2.1 Package definition and import statements

The syntax of the import and package statements in Co-op is very similar to the syntax in Scala. The package declaration in Co-op has the following format:

```
1 package <packagename>;
```

Listing 3.2: A package declaration in Co-op

This will create a Package Declaration element which has the information of the <packagename>. When generating the Scala code, for the package declaration we generate the same line as we had in listing 3.2. Import statements are very similar, they are denoted by the import keyword instead of the package keyword, and generate an Import Declaration as opposed to a Package Declaration.

3.2.2 Main function

If the Co-op program contains a main function we generate a wrapper main function. This wrapper contains functionality to make the arguments passed to the main function into Co-op objects the generated code can handle.

The main function defined in the Co-op code will be generated as a function called $main (see section 3.1.1). This $main function is then called at the end of the main function. The $main function will be generated the same way as other functions (see section 3.2.5 and 3.2.6).

```
1 def main(args: Array[String]) {
2   //transform the arguments into objects that the generated code can handle
3
4   //call the main function ($main) that was defined in the Co-op program
5 }
```

Listing 3.3: The generated main function

3.2.3 Fields

In Co-op we have fields that are members of Co-op classes. For each field we have to generate a getter and a setter. The field is never accessed directly on the variable in the output program that represents the field. This is done because of additional Co-op functionality where we can add annotations on these fields. If we take the field definition as shown in the listing 3.4.

```
1 var Three;
```

Listing 3.4: Example field definition

In the generator we will have an AST element Field Declaration which also contains the name of the field (<fieldname>). For this field we will have to generate a variable, a getter and a setter. For the field we will have to create a variable to hold the value for that field. The type of this variable will be a CoopObject. This variable will be instantiated to our Co-op NULL. As shown in the following listing.

```
1 private var var$<fieldname>: CoopObject = net.sf.co_op.coopiii.lang.Null.NULL;
```

We also will need a getter and a setter for the field as shown in the following listing.

```
1 override def get$<fieldname>(cont : Any => Any) {
2   cont(this.var$<fieldname>);
3 }
4
5 override def set$<fieldname>(value:CoopObject, cont: Any => Any) {
6   this.var$<fieldname> = value;
7   cont(this.var$<fieldname>);
8 }
```

3.2.4 Annotations

Co-op has extensive support for annotations. But most of the annotations are not used yet in Co-op. We decided to not implement support for annotations in this CPS version of Co-op to simplify generation. An exception is the @test annotation, because we need this to run the test cases for Co-op.

The @Test annotation

An annotation we do have support for is the test annotation. This annotation is used to signal that we need to generate a JUnit test. We need to generate a method that JUnit will call when the test case is run. This method has to be in a certain format for JUnit to recognize the method as a valid JUnit test method. The method definition JUnit requires, is in format as shown in listing 3.5.

```
1 @test def <methodname>()
```

Listing 3.5: Method definition for JUnit test method in Scala

For every method in the Co-op program with the test annotation we have to have a method with a signature matching the signature JUnit looks for in a program. This signature is shown in listing 3.5. This signature does not accept the extra continuation parameter. Therefore this method can not be in CPS. We do generate a CPS version of the method with the test annotation but JUnit can not use this method as a test case. In our generated code therefore we have two methods for every test method. First, the method generated in CPS in the same way any other method is generated. Second, the entry point for the JUnit

test, this method is not in CPS, and only responsible for launching the test case. A JUnit helper method has the following definition as shown in listing 3.6.

```
1 @Test def <methodname>$test() {
2   this.<methodname>( (Any : Any) => {
3                                     //Empty continuation
4                                     } );
5 }
```

Listing 3.6: Method definition for JUnit test method in Scala

As shown in listing 3.6 the only thing this method does is call the CPS version of the test method. Because this is the entry point of the program, and the test method is not in CPS, we have no continuation of the rest of the program yet. The argument we pass to the CPS version of test is a new continuation of the rest of the program, which is an empty continuation.

3.2.5 Method definitions

When generating a method we get the AST element Method Declaration that contains the declaration of the method. The declaration contains the parameters of the method. The type of these parameters in the AST is Formal Parameter Declaration, this Formal Parameter Declaration contains the name of the variable. We do not need the type of the parameter, as Co-op is dynamically typed, the type is always CoopObject (See section 4.2).

Because we need the method definition to be in CPS, we always have to add a parameter that accepts a closure with the continuation of the rest of the program. The steps we take to generate this method definition are shown in the following list.

1. Create a method definition with the correct name def <methodname>().
2. For each Formal Parameter Declaration add the parameter, (with type CoopObject) to the method definition.
3. Add the parameter for the continuation (cont : Any => Any)
4. Add this method definition to the CoopObject class (see section 4.2).
5. Add the method definition to the class in which the Method Declaration element is placed. Because this class will inherit from CoopObject, and CoopObject already contains a definition with the same signature, we should add the override keyword.

The following listing shows a method declaration in Co-op.

```
1 method exampleMethod(var aParameter);
```

Listing 3.7: An example Co-op method declaration

The code in the listing above will give us the AST element Method Declaration with the method name exampleMethod. This Method Declaration has one Formal

Parameter Declaration with the name aParameter. If we take this method declaration through the steps for creating the method definition above the following list shows what the result will be.

1. def exampleMethod()
2. def exampleMethod(aParameter: CoopObject)
3. def exampleMethod(aParameter: CoopObject, cont : Any => Any)
4. To CoopObject add the definition
   def exampleMethod(aParameter: CoopObject, cont : Any => Any)
   {
   //Method body for calling static functions (see section 4.2.5)
   }
5. To the class in which the method is declared, add the definition
   override def exampleMethod(aParameter: CoopObject, cont : Any => Any)
   {
   //method body with behaviour of method as defined
   }

3.2.6 Method bodies

When generating method bodies, in the class where the method is defined, we have access to a list of statements the body contains. When generating CPS code for the method body we have to do this in reversed order, as explained below. We need to do this because, when we have a function call in method body, we need to call this function and pass it the continuation of the rest of the program. This continuation consists of two parts, first, the continuation of the rest of the method body we are mapping, and second, the rest of the program after the method. The continuation of the rest of the program we have, as this is passed as an argument to the method (cont) (see section 3.2.5). The continuation of the rest of the method needs to be generated, before we can generate the complete function call.

When mapping the last statement of the function, there are two possibilities: it is a return statement, or it is not. As you know, in CPS there is no "returning" in functions, it continues by invoking the continuation of the rest of the program, or terminates if this continuation is not called. If the last statement of a function is not a return statement, in direct style, the function returns, this is called an implicit return.

Implicit return

If we are mapping a function called m; the caller of m will pass the continuation of the rest of the program as a parameter to m. This continuation is known

inside m as cont. If the last statement in the AST of the method body of m is not a return statement, an implicit return will be appended in the generated code. Because we generate CPS, we do not return, but we execute the rest of the program by invoking the continuation cont. The argument to the invocation of cont should be null. This needs to be done because in the Java code generated by the original Co-op language, we need to always return something because the method has a return type. The methods in the Java version append return null for the implicit returns. Therefore, to have the same behaviour in the Scala CPS version of Co-op we pass the null to the continuation.

Explicit return

If we have an explicit return statement we still have a few options. One, the righthand-side is empty, we simulate this statement the same way we simulate the implicit return, as shown in the following listing.

```
1 cont(null);
```

Two, the righthand-side is a literal. We have to pass this literal <literal> to the continuation, to simulate returning the value, as shown in the following listing

```
1 cont(<literal>);
```

Three, the righthand-side of the return statement is a variable, this the same as if we had literal with the same name, as shown in the listing below.

```
1 cont(<variablename>);
```

Four, the righthand-side contains a function call. We should call the function, the function called is passed the continuation argument. Take the following code of two functions in the same class. exampleFunction returns a literal, and exampleFunction2 has an function call as righthand-side of the return statement.

```
1 method exampleFunction()
2 {
3   return 1;
4 }
5
6 method exampleFunction2()
7 {
8   return this.exampleFunction();
9 }
```

Listing 3.8: An example where the righthand-side of the return statement is a function call

To map return this.exampleFunction(); in the exampleFunction2 in listing 3.8 we call the method exampleFunction and pass the continuation parameter.

The following code is generated for the example in listing 3.8.

```
override def exampleFunction (cont : Any => Any)
{
  cont(new net.sf.co_op.coopiii.lang.Integer(1))
}

override def exampleFunction2 (cont : Any => Any) {
  exampleFunction$1(this, cont)
}
```

Mapping the rest of the method body

After mapping the return statement, we can map the next statement of our method body. In case of implicit return, the last statement, otherwise the one but last statement. The continuation of the rest of the program at this point is no longer the cont parameter passed to the function, it is the rest of the method body.

The following code is the continuation for statements that need to be executed before return this.exampleFunction(); in the body of exampleFunction2, in listing 3.8. (But there are no statements before the return in the example).

```
(cvar: Any) => { exampleFunction$1(this, cont) }
```

We continue mapping the rest of the statements from last to first. The part of the method body already mapped is the continuation for each statement to be mapped.

3.2.7 Assigning to a variable

```
method exampleFunction()
{
  return 1;
}

method exampleFunction2()
{
  var a;
  a = this.exampleFunction()
  return a;
}
```

Listing 3.9: A simple Co-op program containing variable assignment

The line a = this.exampleFunction() in listing 3.9 contains an assignment of the return value of a function. This line calls this.exampleFunction and then assigns the return value to the variable a. We first have to call this.exampleFunction, then assign the "return" value to a and then continue by invoking the continuation of the rest of the program. The return a; statement will be (cvar: Any) => { cont(a) }
The calling of this.exampleFunction, shown in the listing below.

```
exampleFunction(this, <continuation>);
```

continuation consists of two parts, the assignment to a variable a, and the rest of the continuation ((cvar: Any) => { cont(a) }). To assign the return value of exampleFunction to variable a.

```
1 (cvar: Any) => { a = cvar.asInstanceOf[CoopObject]; }
```

This assigns the variable cvar to a. We need to use asInstanceOf[CoopObject] because continuations in our program are of the type Any, but a is of the type CoopObject. cvar will contain the return value of this.exampleFunction. The full continuation of the program when mapping the line 9 will be as shown in the listing below.

```
1 exampleFunction(this, (cvar: Any) => { a = cvar.asInstanceOf[CoopObject]; invokeCont( (cvar: Any)
      => { cont(a) } )});
```

The declaration of the variable a on line 8, still needs to be done before this. We do not have to do any special things to do this in CPS, as this is not a function call. The continuation passed will be wrapped as a continuation again ((Any: Any) => { //continuation }). The full generated cps code for example 3.9 is shown in the listing below.

```
1  override def exampleFunction (cont : Any => Any)
2  {
3    cont(new net.sf.co_op.coopiii.lang.Integer(1))
4  }
5
6  override def exampleFunction2 (cont : Any => Any) {
7    var a : CoopObject = null;
8    invokeCont(
9      exampleFunction(this,
10       (cvar: Any) =>
11         {
12           a = cvar.asInstanceOf[CoopObject];
13           invokeCont(
14             (cvar: Any) => { cont(a) }
15           )
16         }
17       )
18     );
19 }
```

3.2.8 Parameters to functions

If variables or literals get passed as arguments for function calls we do not have to do anything special, we can just pass them to the function.

When we have function calls as parameter for a function we have to keep the execution order in mind. If we have the following code as shown in the listing below

```
1 a(
2   b(
3     d()
4   ),
5   c()
6 )
```

Listing 3.10: Nested function calls

We following is the execution order we expect;

1. d
2. b
3. c
4. a

When mapping these statements to CPS, we must keep this execution order intact. The traversal order we need is called depth first post order traversal. To correctly map function a in listing 3.10 we save the results of the function calls of listing 3.10 in variables. When mapping this statement, we know we have a Method Call AST object. The Method Call object can contain literals, variables, and other method calls as parameters. We start traversing the parameters from left to right, and perform the following steps:

1. If it is literal or variable. Map normally
2. If it is a method call. Replace it with a temporary variable and start mapping this method call at step 1.
3. Otherwise, map the next variable (the one to the right)

This keeps repeating until all parameters are mapped. This (non CPS) code is how we map the function.

```
1 var $tmp1 = d();
2 var $tmp2 = b($tmp1);
3 var $tmp3 = c();
4
5 a($tmp2, $tmp3);
```

Listing 3.11: The (non CPS) code that shows how a function is mapped

The generator will generate the CPS version of the example in listing 3.11. The example in listing 3.11 is not in CPS yet, for explaining, as the CPS version of the code is hard to read.

3.2.9 Conditional expressions

Conditional expressions are not fully implemented. Co-op supports the conditional expression operators && and || for the logical and and or operators. Scala also supports these operators. If we have the following statement a && b with a and b being both expressions, we first have to execute a and the rest would need to be passed as the continuation to the call of a. We could map the conditional expression operators in a couple of steps.

1. Map the left side of the conditional expression operators, and assign the result to a temporary variable (see section 3.2.7).
2. Repeat for right side.

3. Use the Scala conditional expression operators on both temporary variables.

This would be a CPS version of the conditional expressions. But this does have a problem, Co-op has shortcut evaluation, this means that when we have the statement a && b, that if a is false, the result of a && b will always be false, and therefore we must not execute b. If we have a statement in the following form in Co-op, as shown in the listing below.

```
1 if (<expressionA> && <expressionB>) ...
```

Listing 3.12: Possible solution to map the if statement

We can write this (in Co-op) as

```
1 var a = expressionA;
2 var b = false;
3 if (a)
4 {
5   b = <expressionB>;
6 }
7
8 if (a && b) ...
```

Listing 3.13: Co-op code with same semantics as a && b

And this program (as shown in the listing above) we can already map in the generator. Therefore, to map a && b, if we map it in the form of 3.13 we have a correct CPS version.

To simulate the or (||) operator we do not need to execute the right side of the || operator if the left side evaluates to true as we have a || b and a is true, then a || b will always be true. So for the || operator we have to generate the same code as we would for the following Co-op program, shown in the listing below.

```
1 var a = expressionA;
2 var b = false;
3 if (a == false)
4 {
5   b = <expressionB>;
6 }
7
8 if (a || b) ...
```

Listing 3.14: Co-op code with same semantics as a || b

It will only evaluate <expressionB> if <expressionA> evaluates to false. As opposed to the && operator where <expressionB> will only be evaluated if <expressionA> evaluates to true.

3.2.10 If else

If else statements are not fully implemented. The if statement is represented by the If Else Statement in the AST. This If Else Statement consists of:

1. condition This contains the code that determines what code should be executed next.
2. then This contains the code that executes if the condition evaluates to true.
3. else This contains the code that executes if the condition evaluates to false.

The order in which this code should be executed is first the condition, then, based upon whether the condition is true or false, the then or else should be executed. After this it continues by executing the rest of the program.

The condition needs to be executed first, and it can contain function calls, therefore, we can not simply map the statement

```
1 if(<condition>)
2 {
3   //mapped version of "then", followed by the continuation of rest of the program
4 }
5 else
6 {
7   //mapped version of "else", followed by the continuation of rest of the program
8 }
```

Listing 3.15: Possible solution to map the if statement

If <condition> would be a variable, then this would be valid CPS. But if the <condition> contains, or is, a function call the program will not be in CPS. Therefore we first have to evaluate the condition expression, and assign the result of the expression to a temporary variable (see section 3.2.7). In listing 3.15 we have to map the statements for then and else and invoke the continuation of the rest of the program there. Otherwise, if we would invoke the continuation after the complete If Then Else construct we would generate invalid CPS if the then or else would contain function calls.

3.2.11 Loops

Loops are not fully implemented. When implementing loops in CPS we come across a problem. The following listing shows an example of the while loop in Co-op.

```
1 var a;
2 a = 0;
3 while (a < 10)
4 {
5   //while loop body
6   a = a + 1;
7 }
```

Listing 3.16: Example of a loop in Co-op

This code in the listing above is valid in CPS as there are no return statements. But when we have a method call in the body of this loop, (line 5) we can not generate it the same way. If we have a method call to a method m in the body of the loop, this could not be done in CPS, when using the standard loop

constructs. We would have to pass the continuation of the rest of the program as a parameter to the call of the method m. This continuation would have to contain the information on how to execute the rest of the loop, but we can not pass this information. Therefore, we need to simulate the loops, while not using the loops in the generated language (Scala).

In Co-op we support three different loops, for, while, do while.

for loop

The for loop has the following syntax:

```
ForLoopStatement:
  'for' '(' (initialization=AssignmentExpression)? ';' (condition=AssignmentExpression)? ';' (increment=
      AssignmentExpression)? ')' statement=Statement
;
```

Listing 3.17: The syntax of the for loop in Co-op

The for loop contains four parts:

1. initialization
   We can initialize the variable for the loop condition here. Co-op does not support the creation of a variable in the initialization that would have the scope of the for loop.
2. condition
   The condition to check and see if we need to continue the loop or that we are done.
3. increment
   The increment is executed after the statement
4. statement
   The code inside the loop can be a block of statements.

We need to simulate the for loop statement in the generated Co-op code, and we want all the elements of the loop to be first class so we can pass them around. We can do this by, when mapping a for loop, calling a function that simulates the for loop, while passing all the elements of the loop as closures. Before calling the function, we can map the initialization statement, as this needs to be done in the beginning of the loop. We can consider the initialization as a statement before the loop and can map it as such. Then we map the rest of the for loop as a call to a function with the following definition.

```
def forLoopFunction(condition: (Any : Any), increment: Any => Any, statement: Any => Any, cont:
    Any => Any) {
  // map the body of the for loop
}
```

Listing 3.18: The definition of the foor loop function

The body of the forLoopFunction should simulate the for loop while being in CPS.

```
1  def forLoopFunction(condition: (Any : Any), increment: Any => Any, statement: Any => Any, cont:
       Any => Any) {
2  if (condition)
3  {
4    <statement> //execute the body of the loop
5    <increment> //increment the loop
6    forLoopFunction(condition, increment, statement, cont);
7  }
8  else
9  {
10   //loop is done, continue by invoking the continuation of the rest of the program
11   cont();
12 }
```

Listing 3.19: How the for loop as a function should work in Co-op

We can already map the if statement to CPS (see section 3.2.10). Lines 3-5 of listing 3.19 we still have to map. We can map these lines the same way we would map a method body (see section 3.2.6). The only difference is that we do not need to map implicit returns, as we always finish the statements by calling the forLoopFunction which is responsible for executing the rest of the program. This forLoopFunction will be uniquely generated for every for loop used in the Co-op program.

while loop

The while loop has the following syntax:

```
1  WhileLoopStatement:
2    'while' '(' condition=AssignmentExpression ')' statement=Statement
3  ;
```

Listing 3.20: The syntax of the while loop in Co-op

The function we will have to call that simulates the while statement, continuously executes statement until the condition no longer holds, by calling itself.

```
1  def whileLoopFunction(condition: (Any : Any), statement: Any => Any, cont: Any => Any) {
2  if (condition)
3  {
4    <statement> //execute the body of the loop
5    whileLoopFunction(condition, statement, cont);
6  }
7  else
8  {
9    //loop is done, continue by invoking the continuation of the rest of the program
10   cont();
11 }
```

Listing 3.21: How to the while loop as a function should work in Co-op

Again we would map line 3-4 in the listing 3.21 the same way we would map a method body.

dowhile loop

The dowhile loop has the following syntax:

```
DoWhileLoopStatement:
  'do' statement=Statement 'while' '(' condition=AssignmentExpression ')' ';'
;
```

Listing 3.22: The syntax of the dowhile loop in Co-op

The do while loop is very similar to the while loop (see section3.2.11). The only difference is that we have to execute the statements first, and checking the condition after.

```
def dowhileLoopFunction(condition: (Any : Any), statement: Any => Any, cont: Any => Any) {
  <statement> //execute the body of the loop
  if (condition)
  {
    dowhileLoopFunction(condition, statement, cont)
  }
  else
  {
    cont();
  }
}
```

Listing 3.23: How to do while loop as a function should work in Co-op

The <statement> in the listing above, again is mapped as a method body (see section 3.2.6).

We now know how we should map the three loops used in Co-op. It requires unique functions to be generated for every loop in the Co-op program, so for every loop in the program there will be a function generated that simulates that specific loop. The loops in Co-op support the break and continue statements, which we will map next.

break

The break keyword is used to exit loops early. While in a function that simulates the loops (see sections 3.19, 3.21 and 3.21), we need to simulate this by executing the rest of the code that would come after the loop. For all three loops this can be done by invoking cont, the argument given to the loop function, which contains the continuation of the rest of the program.

continue

The continue keyword is used to skip the remainder of the current iteration, and continue with the next iteration. In the functions we use to simulate the loops, we can do this by calling the function of the loop we are currently in. If one of the statements in the <statement>, block for example in listing 3.21 would

contain the continue keyword, we can simulate this by calling dowhileLoopFunction(condition, statement, cont).

3.2.12 operators

Some operatorsb such as the + and the - operator, are translated to function calls in Co-op. We can use these operators on classes we defined ourself by having the function in the class with the name corresponding to the the operator.

Table 3.1: Co-op operators and their corresponding functions

Operator	**function**
+	plus
-	minus
*	multiply
/	divide
%	mod
&	bitwiseAnd
\| | inclusiveOr |
^ | exclusiveOr |
== | equals |
!= | notEquals |
< | lessThan |
> | greaterThan |
<= | lessThanOrEquals |
>= | greaterThanOrEquals |
<< | shiftLeft |
>> | shiftRight |
>>> | unsignedShiftRight |
? | condition |
[| get |

When mapping an operator from the table 3.1, we can map them to corresponding function. The code in Co-op:

```
a <operator> b
```

will be translated to:

```
a.<function>(b)
```

The classes from the standard Co-op library (see section 4.2.4) already contain the functions for (some of) these operators. This ensures we can use, for example, the + operator on two integers in Co-op. By implementing functions with the function name in table 3.1, Co-op users can use the operator for classes that support them.

3.2.13 throw

Co-op supports the throw keyword to throw an exception. It requires a throwable object as an argument. Since Co-op does not support the catching of exceptions, we do not have control over how the exception is handled. So we just throw the exception because we can not control it anyway.

If Co-op later were to support the catching of exceptions, then just throwing the exception as we do now would not be enough. We would have to create a wrapper for exceptions, that contains the exception, and the continuation of the execution of the rest of the program.

Chapter 4

Invoking methods that are not in CPS from CPS code

One of the requirements for us is to be able to call functions defined in Java libraries, in Co-op (see section 1.3.3). For example, we want to be able to use the functionality Java libraries provide, such as the collection classes. We also want to be able to use third party libraries written in Java. Being able to use functionality defined in existing libraries, will make programming in Co-op easier.

We want our generated code to be in CPS. Having the generated code in CPS will require all the function calls in CPS. The signatures of the functions defined in the Java libraries are not in CPS, meaning that the functions do not accept the extra continuation parameter nor invoke this continuation when the function is done. Also, these functions return. Because we do not have control over the libraries we want to use, we can not change the functions in the libraries to CPS. We therefore have to find a way to be able to interact with these library functions, which are not in CPS, in our generated code which is in CPS.

To find a way to call these functions we have to investigate how these function calls are done in Co-op. First, in section 4.1 we list what types of function calls we are going investigate. And see what code at the call site in Co-op we need to call these functions types. Then, in section 4.2 we are going to investigate how these types of function calls are invoked in Co-op, and what parts of Co-op are involved in this. Finally in section 4.3 we are going to see what we need to do to call functions, from the types of functions described, in the CPS version of the Co-op language.

4.1 Types of function calls

We distinguish function types on a number of characteristics. The language of the call site of the function, the language for the target of the function call. These characteristics determine over what part of the process we have control. As we have control over the Co-op code, but not over the Java code. Another characteristic is if the functions are static or not. This determines how the call site is, and if we call the method on an instance of a class. For the relevant function types we will explain in later subsections, how the calling of these types of functions works at the call site in the Co-op language, and how they are integrated in Co-op.

Table 4.1: Characteristics for function types

function types	**call site**	**target**	**static**	**relevant**
Functions written in Co-op	Co-op	Co-op	no	yes
Static Co-op functions	Co-op	Co-op	yes	no
Non static Java functions	Co-op	Java	no	yes
Static Java functions	Co-op	Java	yes	yes
Java functions called from Java	Java	Java	no	no
Java static functions called from Java	Java	Java	yes	no
Co-op functions called Java	Java	Co-op	no	no
Static Co-op functions called from Java	Java	Co-op	yes	no

Functions written in Co-op. The first function type are the non static functions, written in Co-op. These are all in direct style. We have control over how we call these methods in Co-op as well as the signature of these methods. They are defined in direct style, but the methods need to be in CPS, and have a CPS signature. Therefore we have to transform the methods. We also can transform all the method calls so they call the methods in CPS, passing the current continuation as an argument. We only transform the method signature, and the signature of the call. Because we do not have to transform how functions are invoked, we do not care if the function, overridden. Because overridden functions already work in the original Co-op code, we do not have to take this into account.

Static Co-op functions. In Co-op it is not possible to define static functions, therefore, we do not have to take this type of function into account.

Non static Java functions. Non static Functions in libraries written in Java is another type. We have control over how we call these methods, but not on their signature. Because we call these methods on instances of the Java object that contains the function, we do not need to worry about how the method is

dispatched. We can handle all non static functions calls to libraries in Java the same way.

Static Java functions. The call site for static Java functions is different than from calling non static Java functions, as you do not call the method on an instance of a class. We need to call static Java functions from Co-op. Therefore, Static Java functions is a different function type to consider.

Java functions in Java. We do not need to consider these type of functions as we have no control over them.

Java static functions in Java. We do not need to consider these type of functions as we have no control over them.

Co-op functions from Java. An example of when a Java function would call a function in a Co-op class is when the Java code does a callback to code in Co-op. We have not considered this type of function, as do not have any scenarios using this yet.

Static Co-op functions from Java. As in Co-op we can not define static functions, therefore, we do not have to take this type of function into account.

4.1.1 Functions written in Co-op

We have total control over how these functions are generated. Listing 4.1 is an example of a definition and call to a native Co-op function. We define the function getNumber (line 3). The function does not specify a return type as it is not possible to specify a return type in Co-op because they always have the implicit return type CoopObject. Functions can return values (as we do on line 4) because the generated function returns a CoopObject.

```
1  class CoopExampleClass {
2
3    method getNumber() {
4      return 1;
5    }
6
7    method testCoopMethod() {
8      var number;
9      number = this.getNumber();
10   }
11
12 }
```

Listing 4.1: The definition and calling of functions written in Co-op

4.1.2 Java non-static functions

Java non-static functions are called on instances of Java classes. We might not have any control over these classes, and can not generate a CPS version of them. An example is the HashMap. When Java library class HashMap is imported in a Co-op file, we can use this class as we would a class that is defined in Co-op. We can instantiate the class by calling HashMap.new(), and can then assign this instance to a variable. We then can call the methods the HashMap class provides on this variable, as shown in the listing below.

```
1  import java.util.HashMap;
2
3  class UseMap {
4
5    method useJavaMap() {
6      var doubles;
7      doubles = HashMap.new();
8
9      doubles.put(1, 2);
10   }
11 }
```

Listing 4.2: Calling a non-static Java function

4.1.3 Static Java functions

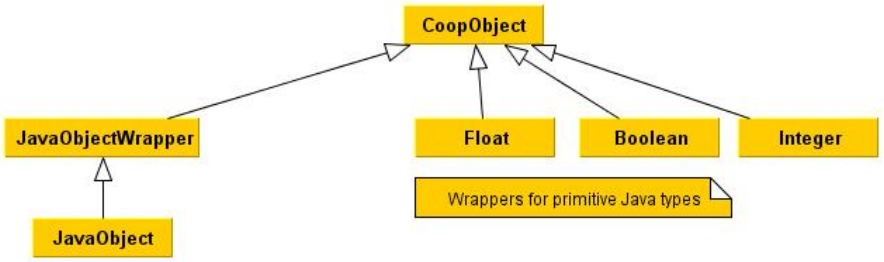
Static methods can not be defined in Co-op, but we are able to call static methods defined in Java classes, from Co-op. The call site in Co-op is different from how you call normal Java functions in Co-op. In Java, we do not need to instantiate a class, but can call the methods directly. In Co-op however, we create an object on which to do the static call when generating (as shown in section 4.2.5). Below is an example of the static call assertEquals on the Assert class.

4.2 How method calls are dispatched in Co-op

Because Co-op is a dynamically typed language, but Java is statically typed we have make sure generated code in Java can resolve the method calls to compile and call methods. Co-op uses a base type, CoopObject, as the base type for all types. When a variable is declared in Co-op, the only thing we can be sure of is that the base type is this CoopObject. CoopObject is not responsible for the implementation of methods, the class in which the method is defined in Co-op is. And when using Java types we have another problem, the Java types are not subclasses of CoopObject. We therefore, have to ensure that Java compiler can resolve the method calls, and ensure they can be done, and at runtime, the method is invoked on the correct object.

Figure 4.1 contains the elements, involved in Co-op method invocation, that we explain in the following sections. CoopObject is the base type for all types used. JavaObjectWrapper and JavaObject are used for Java integration. The wrappers for primitive types (Float, Boolean, and Integer) are classes used in the runtime Co-op library for using primitive types.

Figure 4.1: Class diagram of the Co-op classes used for function calling



We have classes that are part of the Co-op runtime library, they do not change each time we generate Co-op code. And we have classes that are compiled specifically for each Co-op program. The classes that are generated each time contain method definitions for all methods called in the Co-op program compiled. They are added to ensure the function exists when compiling as we will see later. This table shows what classes are generated differently for each Co-op program, and which are not.

Table 4.2: Classes used and whether they are different for each Co-op program

Class	**Generated differently for each Co-op program**
CoopObject	Yes
JavaObject	Yes
JavaObjectWrapper	No
Primitives (Integer, Boolean, Float)	No

Explanation for table 4.3. Each function is a member of an object, the compile time type of this object determines how the Java compiler statically links the functions calls in our compiled Co-op program. Because Co-op is a dynamically typed language, while Java is statically typed. Because we compile to Scala the compiler should be able to link the function calls to assure they can be done. At runtime the method can be invoked on a subtype of the compile time type. Because the type on which the function is invoked at runtime is different for the types of functions, we can use this to distinguish between the function types. The type on which the call is invoked at runtime will contain the code for calling

the function, having the knowledge of what type of function call is done. Each row in table 4.3 is explained in a separate subsection of section 4.2.5.

Table 4.3: Types of function calls and their compile and runtime types

Type of function call	**compile time type**	**Type on which the call is invoked at runtime**	**comment**
Functions written in Co-op	CoopObject	The object in which the called function is defined	The subtype of CoopObject that is instantiated
Java non-static functions	CoopObject	JavaObject	Invokes the called method on a instance of the Java class
Static Java functions	CoopObject	CoopObject	CoopObject contains code for invoking the static method

4.2.1 CoopObject

Co-op is a dynamically typed language, but Java is a statically typed language so we need a base class to act as the type of all variables. In Co-op this base class is CoopObject, every object in the generated Co-op program is a (sub)type of CoopObject.

When generating the Java code, a method is added to CoopObject for every method called in the program, including the methods on Java libraries. Because objects are always of the type CoopObject, and CoopObject contains a method with the correct signature, at compile time (see table 4.3), the Java compiler can always ensure that a method call can be done.

As shown in table 4.3, only Static Java functions have the runtime type of CoopObject. If at runtime a method is called on CoopObject, we can execute the behaviour for invoking static function calls.

4.2.2 JavaObjectWrapper

The JavaObjectWrapper is used for integration with Java classes. Because the Java objects from libraries do not inherit from CoopObject we have a wrapper for them. The JavaObjectWrapper is this wrapper, it inherits from CoopObject, and also it contains a reference to the Java object it is wrapping. This enables us to use Java classes, while the compile time type is CoopObject (see table 4.3). JavaObjectWrapper also contains some important functions that help us handle Java objects.

It contains the unwrap method that returns a JavaObject (see section 4.2.3) if the argument is a Java object otherwise it retuns a CoopObject (see section 4.2.1). This method is used to, at runtime, differentiate between objects of types that are created in Co-op, and objects that are part of a Java library. Because unwrap returns a different type, for Co-op classes and Java classes, when invoking the functions, we know that if a method in JavaObject is called, it is a method that is part of a Java library.

JavaObjectWrapper also contains the invoke method. invoke is the function that calls the method, that is an argument for the invoke function, on the Java object it wraps. It uses reflection to find the Java method that needs to be called. invoke is the method responsible for invoking all the Java non-static methods. The method wrappers generated in JavaObject (method calls that should be done on a Java library) use this function.

4.2.3 JavaObject

JavaObject is a subtype of JavaObjectWrapper, that is generated each time a Co-op program compiles (see table 4.2). JavaObject contains a method definition for all functions called in the Co-op program. Not only the Java functions, because we cannot distinguish between the types of calls in the generator while generating the JavaObject class. Because the JavaObjectWrapper.invoke function only returns a JavaObject when we need to call a Java class, we know that when a method is called on JavaObject, it is a method defined in a Java library (see section 4.2.2). The bodies of the methods defined in JavaObject call the invoke function, defined in JavaObjectWrapper to call the methods in the Java library used.

4.2.4 Primitives

As Co-op is dynamically typed, and we use CoopObject as a base type, everything is a CoopObject. In Java there are primitive types, such as int. These primitives should also become objects in Co-op, so they can have the supertype CoopObject. The Co-op runtime library contains wrappers for some of the primitive types, currently Integer, Boolean, and Float. In code that is generated by Co-op, integers, for example, are not of the primitive type int or the Java object Integer, but they are Co-op objects of the type net.sf.co_op.coopiii.lang.Integer.

If we for example have the following line, where we use an integer in a Co-op program:

```
1 return 1;
```

This will generate an instance of the Co-op wrapper for Integer. As shown in the next listing.

```
1 return new net.sf.co_op.coopiii.lang.Integer(1)
```

The Co-op generator translates binary operators, such as the + operator, to function calls. The Co-op primitive classes implement methods for these binary operators, for example, there is a plus function that is used to implement the + operator.

4.2.5 Calling functions in the original Co-op implementation

In this section we will summarize how examples of the three types of function calls in the section 4.1 are handled, using the concepts explained in section 4.2. The original Co-op implementation is the version of Co-op that we are changing to CPS. We explain it using the original Co-op implementation to see what we need to do to make the calling of functions possible in the CPS version of Co-op.

Functions written in Co-op

```
1  class CoopExampleClass {
2
3    method getNumber() {
4      return 1;
5    }
6
7    method testCoopMethod() {
8      var number;
9      number = this.getNumber();
10   }
11
12 }
```

The class CoopExampleClass has the supertype CoopObject. this on line 9 has, at compile time, the type CoopObject (see table 4.3). Because CoopObject contains a method for all calls made in the Co-op program it will also contain getNumber. When generating Java code the compiler knows we want to execute the getNumber function on a CoopObject. The version of the getNumber function will, as we have seen in section 4.2.1, contain how we should execute a static function with that name. The runtime type of this will be CoopExampleClass (See table 4.3). The CoopExampleClass class overrides the getNumber function. CoopExampleClass.getNumber contains the method that should be invoked with the generated code to simulate return 1;. Because the runtime type of this will be CoopExampleClass (see table 4.3), the CoopExampleClass.getNumber method is invoked.

Java non-static functions

```
1  import java.util.HashMap;
2
3  class UseMap {
4
5    method useJavaMap() {
6      var doubles;
7      doubles = HashMap.new();
```

```
8
9       doubles.put(1, 2);
10    }
11  }
```

We have a variable called doubles to which is assigned an instance of a java.util.HashMap. At runtime, by using the JavaObjectWrapper.unwrap method (see 4.2.2), the runtime type of this variable will be a JavaObject (see table 4.3). The JavaObject contains the definition for put as follows.

```
1  public CoopObject put(CoopObject arg1,CoopObject arg2) {
2    return invoke("put",arg1,arg2);
3  }
```

This calls the JavaObjectWrapper.invoke (see section 4.2.2) method that will invoke the put method with arguments arg1 and arg2 (the parameters 1 and 2 in our example) on the object it wraps (a java.util.HashMap)

Static Java functions

```
1  import org.junit.Assert;
2
3  class ExampleCallStatic{
4
5    method callStatic(){
6      Assert.assertEquals(4, 2 + 2);
7    }
8  }
```

When calling the static function assertEquals on the Assert class, the Co-op compiler will create a CoopObject for Assert. Because it is not unwrapped to a JavaObject by JavaObjectWrapper.unwrap it will be a CoopObject at runtime (see table 4.3). Unlike functions written in Co-op, the method assertEquals is not overridden in another class. So, at runtime, when Co-op calls assertEquals, the implementation of the CoopObject will be invoked. The assertEquals method in CoopObject will contain code to call JavaObjectWrapper.invoke to invoke the static method.

4.3 Calling functions in the CPS Co-op implementation

Now we know how calling of the three different method types works in the original Co-op implementation, we are going to see what adjustments we have to make to change the generation to CPS.

4.3.1 Functions written in Co-op

For functions written in Co-op the invocation does not change. All method signatures and calls will be in CPS. The method calling will not change.

4.3.2 Java non-static functions

We can not call the Java non-static functions with a CPS signature, as they do not have a CPS signature. Therefore, we have to create a bridge between the CPS code and the direct style function in the Java library.

One of the solutions is to call the function in the Java library in direct style. When generating code for the function call (see Chapter: 3 Changing the generator) we could just have the call in normal style, as opposed to CPS.

This has a some disadvantages. While debugging the generator, looking at generated Co-op classes, we will see mixed CPS and direct style in function bodies. This will make it harder to debug since the code will be less readable. Also, when mapping the function call in the Co-op generator, we have to distinguish between calls to functions written in Co-op and Java non static functions. This would lead to different code generations for the same construct in the Co-op call site. We can not distinguish between Java and Co-op functions at the generator. To be able to distinguish, we would have to add information to the method declaration object in the AST. But we already decided that if possible, we do not want to change the AST, only the generator.

As we have seen in this chapter, all calls to Java non-static functions are called via JavaObject. If we keep the signature of this method to be in CPS, then we can handle the generation for the calling of the functions the same as functions written in CPS. The following listing shows an example of CPS wrapper in JavaObject

```
1 override def put(CoopObject arg1, CoopObject arg2, cont : Any => Any) {
2   var t = invoke("put",arg1,arg2);
3   cont(t);
4 }
```

Listing 4.3: The functions of JavaObject in the CPS version of Co-op

This function has a CPS signature, but it is not in CPS, neither is the invoke function.

We chose to do the bridge between CPS and direct style in the JavaObject rather than also change the invoke method in JavaObjectWrapper. We did this because of maintainability. Currently we maintain both the original Co-op version and the CPS version. Because we do not change JavaObjectWrapper we can a use new version of the Co-op runtime library from the original Co-op version, without having to manually change it to CPS. If we only would maintain the CPS version, it would be better to change the invoke method to be in CPS, as this way invoking a Java function would take one less direct style function call.

4.3.3 Static Java functions

As when calling the Java non-static functions, we only want to change the part where we actually call the function. In the case of static Java functions this happens in CoopObject. We change the signature of the method in CoopObject to CPS, but the method body is in direct style, invoking the static call. The following listing shows the code generated for functions in CoopObject

```
1  def assertEquals(arg1: CoopObject, arg2: CoopObject, cont : Any => Any) {
2    try
3    {
4      var javaClass = this.asInstanceOf[net.sf.co_op.coopiii.lang.Class].toJavaClass();
5      var x = JavaObjectWrapper.invoke(javaClass, "assertEquals", null,arg1,arg2);
6      cont(x);
7    }
8    catch
9    {
10     case e: Exception => throw new Error(e);
11   }
12 }
```

Listing 4.4: The functions of CoopObject in the CPS version of Co-op

Again we could also have the method in CoopObject be fully CPS, and create a CPS version of JavaObjectWrapper.invoke. But we did not do this as we can now use newer version of JavaObjectWrapper without having to change them.

Chapter 5

Evaluation & Conclusion

In this chapter we will evaluate how we created the CPS version of Co-op. We are going to evaluate the decisions we made during the development.

5.1 Using Scala with closures

Using the Scala and the closures we were able to create a CPS version of Co-op that satisfied our requirements (see section 2.3). To evaluate this decision we are going to look at the requirements we listed in chapter 2.3, as well as readability and ease of use. We also evaluate if using Scala with closures was a correct decision by answering the question "If we have to make the same choice again, would we still take Scala with closures?".

5.1.1 Requirements

In chapter 2 we defined requirements for the language and technique we used for the new CPS version of Scala. We are going to evaluate for each requirement if using Scala with closures worked out as expected (See section 2.6.2).

R1. Must be possible to output CPS

The new version of Co-op that outputs Scala code in CPS works. As we have all the generation strategies necessary to create a CPS version in chapter 3, we know using Scala with closures we are able to create a CPS version of the Co-op language, without having the CPS version finished.

R2. Backwards compatibility with the Original Co-op implementation

Because of the compatibility between Scala and Java we are able to use Java objects and invoke Java methods using Scala. It requires no extra code or external libraries and is an excellent way to interact with external Java code as described in chapter 2.

R3. Support in Eclipse

The Scala Eclipse plugin made it easy for us to develop Scala within the Eclipse IDE. During the development of the CPS version of Co-op we used the plugin for its syntax highlighting, code completion, and the debugger and compiler. Syntax highlighting helped us to easily identify problems with generation and provided readability. The code completion and debugger allowed us to create test programs to see what output we wanted to generate and test this before changing the generator. The integration of the compiler with eclipse allows us to automatically generate new output when changes are made to source files. In conclusion the support in Eclipse was excellent and it helped us develop the CPS version of Co-op.

R4. Ease of development

Ease of development is harder to specify. The closures in Scala are easy to create. During development we had no problems with the closures that Scala provides, however, it was not easy to develop the CPS version of Scala. We expect that the difficulties we had while developing where inherent to CPS, and not specific to using Scala with Closures. However, we do not know this for sure, as we only created a CPS version using Scala with closures.

5.1.2 Lessons learnt

In this section we will evaluate other consequences of our choice to use Scala with closures that we did not discuss in the requirements section. These lessons learnt are important to answer the question: "If we have to make the same choice again, would we still take Scala with closures?".

Readability

The readability of the generated code is not optimal.

It is hard read the code generated code. The fact that the code is hard to read however is mostly due to the fact that it is in CPS. CPS is inherently harder

to read as passing the continuation of the rest of the program to each function call makes it harder to understand as the code is less structured than it would be in normal style.

While the generated code is hard to read, this is not due to the fact that we chose Scala with closures as our language and technique to use. The closures are really direct and do not provide a lot of clutter to create. As an example we can create a closure in Scala using the code shown in the following listing.

```
1 (Any : Any) => { //code in the closure }
```

Listing 5.1: An example of a closure in Scala

We only need to specify a possible input and output type, and the rest is just brackets to envelop the code. This amount of code necessary to create closures is less then when we would use the alternatives. Using Java we would have to create an instance of an anonymous class, with a function definition in the anonymous class definition. And then we would still need to add some behaviour to be able to access the necessary variables (see 2.5.1). Using continuations in Scala (see 2.6.1) we would have a lot of nested shift and reset statements, providing extra code, and less readability as you would always have to look what shift and reset belong together.

Ease of use

The development of the CPS version of Co-op had some difficulties. The generator was hard to debug, the timespan form adjusting a line in the generator to see the result in the generated code took a long time, as well as some other problems changing the generator. All of these problems, however, were not related to using Scala with Closures, but were inherent to the techniques used to create the Co-op generator, namely XText and XPand. The ease of use of Scala in Eclipse was excellent. It provides a compiler and debugger, integrated in Eclipse as well as syntax highlighting and code completion. All in all we can conclude Scala with Closures is easy to use in Eclipse, even tough the development process was rough.

5.1.3 If we have to make the same choice again, would we still take Scala with closures?

Yes, under the same circumstances we would. We concluded by going over the requirements that the assumptions we made about using Scala with closures were correct, and we did not find any factors outside the requirements that would prevent using Scala with closures as the language and technique used to create the CPS version of Co-op.

The only other viable alternative language and technique to use was Java with function objects (see chapter 2). An advantage we would have using Java is we

already had an implementation of Co-op that generated Java code. As described in section 2.2 we altered this version to generate Scala and from there started to create the CPS version. As we have seen in the requirements en lessons learnt sections, we did not find any reasons not to use Scala with closures. Therefore, if we would have to make the choice again, we would have to weigh the initial cost of changing the generator to Scala, and using closures, against using Java with function objects. We expect that using Java with function objects would be more difficult because the functions objects do not provide the same functionality as closures (see section 2.5.1).

Therefore, if we have to make the same choice again, under the same circumstances we would again choose to use Scala with closures. The initial cost of creating the direct style Scala version of Co-op would be less then the extra work we would have to do to get the same functionality as closures using function objects. The experience we had using Scala with regards to the backwards compatibility, support in Eclipse, readability, and ease of use are positive. Therefore we would also not have to look for another alternative if we had to make the same choice again, but choose for Scala with closures.

5.2 Invoking methods that are not in CPS from CPS code

To evaluate our solution for Invoking methods that are not in CPS from CPS code (chapter 4), we are going to evaluate the technique we used on three points

- Ease of generation
  How much does the fact that some methods need to be called differently because they are not in CPS (see chapter 4) influence how we generate code?
- How close to pure CPS
  We evaluate how close to pure CPS our chosen approach is, how much of the generated code is not in CPS, and how much time in execution the program is not in CPS?
- Dead code
  How much dead code is generated to make the different types of method calls possible?

These items are discussed in the following subsections.

5.2.1 Ease of generation

Because of how Co-op method invocation works, it is easy to do the generation. At generation, each of the type of function calls we considered (see section 4.1)

has a different runtime type. We generate the code that handles the method invocation for each method type, for all functions. At runtime the correct implementation will be called based on the runtime type. This makes the generation of code easy, as we do not have to distinguish between the types of function calls when generating the code.

5.2.2 How close to pure CPS

All of the method calls have CPS signatures. In the generated code, everything except for the code in the wrappers for calling the non CPS functions, is in CPS. Because of this, when executing the program, the only time the execution is not in CPS is in the wrappers specific for calling non CPS code.

5.2.3 Dead code

For every method call, we generate code to handle the method call, but as we do not know the type during the generation, we generate the code to call the function for every method type (see chapter 4). Therefore, there is code in the program that will never be executed. This has the disadvantage that a lot of dead code is generated. Which is bad for the code quality and readability.

5.2.4 If we were to make the same choice again, would we still take the same approach?

Yes, the changes to the generator are easy and we do not need to distinguish between the different method types the function calls are (see section 4.2.5). This makes the generator itself easy but less clean. The code is also close to pure CPS, as the execution will only be not be in CPS if a method is called that is not in CPS. The generated code is not easy to debug, but this is a problem that is inherent to the approach of function calling in Co-op. All in all we are satisfied with the approach we took and have not found any problems with it.

5.2.5 Tail call optimization

A point to keep in mind with the generation of CPS code is that the stack may grow infinitely. We did not consider this as in our test cases we never came across this problem but it might occur when programs get larger. The problem is that as CPS code never returns, the stack will keep growing during the execution of the program. Tail call optimization [14] is a solution where stack frames for functions will be discarded if we would never need them again. This is a principle that would have to be supported in the generated language. The Java VM does not support this. Because we call functions in Java, that are

not in CPS, you can not discard all old stack frames as the Java functions do need these when returning. Generating CPS loops as described in 3.2.11 creates a lot of function calls so this leads to a large stack and then a stack overflow can occur. If someone wanted to create CPS programs using the techniques we described, we advise to look at tail call optimization as otherwise stack overflows might occur in larger programs where there are a lot of function calls.

5.3 Generator

Co-op has a set of test cases designed to test the generator. These test cases are designed to test specific parts of the generator. These test cases test specific techniques that need to be generated. Table 5.1 contains a list of techniques for the generator (see chapter 3), which test cases use these techniques, and if we have a generation strategy.

Table 5.1: Other used techniques in Co-op

Technique	**Test cases using the technique**	**Generation strategy?**
package definition and import statements	Annotations, Classes, Cps, Factorial, Greatestnumber, Hello, UseMap, Keywords, Loops, MyInteger, Person, PrimeNumber, Primitives, DifficultSyntax, Three	Yes
main function	Factorial, Hello	Yes
fields	MyInteger, PrimeNumber, Primitives	Yes
annotations	Annotations	No
the @Test annotation	Annotations, Classes, Cps, Factorial, Greatestnumber, Hello, UseMap, Keywords, Loops, MyInteger, Person, PrimeNumber, Primitives, DifficultSyntax, Three	Yes
method definitions	Annotations, Classes, Cps, Factorial, Greatestnumber, Hello, UseMap, Keywords, Loops, MyInteger, Person, PrimeNumber, Primitives, DifficultSyntax, Three	Yes
method bodies	Annotations, Classes, Cps, Factorial, Greatestnumber, Hello, UseMap, Keywords, Loops, MyInteger, Person, PrimeNumber, Primitives, DifficultSyntax, Three	Yes
implicit return	Annotations, Classes, Cps, Factorial, Greatestnumber, Hello, UseMap, Keywords, Loops, MyInteger, Person, PrimeNumber, Primitives, DifficultSyntax, Three	Yes
explicit return	Annotations, Cps, Factorial, Greatestnumber, Hello, MyInteger, Person, PrimeNumber	Yes
assigning to a variable	Annotations, Classes, Cps, Factorial, Greatestnumber, Hello, UseMap, Keywords, Loops, MyInteger, Person, PrimeNumber	Yes
parameters to functions	Annotations, Classes, Cps, Factorial, Greatestnumber, Hello, UseMap, Keywords, Loops, MyInteger, Person, PrimeNumber	Yes
conditional expressions	Factorial, Loops, PrimeNumber	Yes
if else	Factorial, Greatestnumber, PrimeNumber	Yes
for loop	Loops, PrimeNumber	Yes
while loop	Loops	Yes
dowhile loop	Loops	Yes
break	Loops	Yes
continue	Loops	Yes
operators	Greatestnumber, Loops, MyInteger, Person, PrimeNumber, Primitives, Three	Yes
throw		No

5.3.1 Test cases

The table 5.2 contains all test cases with a description, it also contains columns that show if the current version of the Co-op compiler can successfully compile and run the test cases. It also lists if we have the generation strategy needed to implement the necessary technique to successfully compile the test case.

Table 5.2: Test cases for the Co-op generator

Test case name	**Description**	**Automated test case runs?**	**Generation strategy available**
annotations	Testing annotations	No	No
classes	Testing the construction of classes	No	Yes
cps	CPS specific tests	Yes	Yes
factorial	Calculating a factorial with recursion	No	Yes
greatestnumber	Find greatest number	No	Yes
hello	Print a string	No	Yes
intergration	Test a Hashmap	No	Yes
keywords	Test if using certain keywords can break compilation	Yes	Yes
loops	Testing loop constructs	No	Yes
myint	Integer tests	No	Yes
person	A test person class	No	Yes
primenumber	Calculate if a number is prime	No	Yes
primitives	Primitive testing	No	Yes
syntax	Miscellaneous syntax tests	No	Yes
three	Simple test class	No	Yes

Because the generator is not complete yet, and there are still some bugs with the basic techniques such as "parameters to functions" that result in invalid syntax being generated. However, we do have generation strategies for all test cases except the Annotations test case, which we decided not to handle (see section 3.2.4). Therefore, we conclude that if all the proposed generation strategies that are described in chapter 3 would be implemented, we would have a fully working CPS version of the Co-op compiler, that generates Scala code with closures.

5.3.2 Ease of generation

The changes to the generator, as described in chapter 3, make the generator more complex. A lot of constructs have one generation strategy in the original compiler that compiled to Java code in direct style. In the CPS version we have

more strategies for generating the same element based on the context in which the element is placed. Therefore, the generator for the CPS version of Co-op is much more complex than the original Co-op version. The generator is harder to maintain, it is more prone to bugs because of the complexity.

5.3.3 Generated code

The generated code is used while debugging programs written in Co-op. The readability of the code generated is not optimal. This is mostly due to the fact that CPS code is hard to read. There is not much clutter in the code for creating and using the closures used to implement CPS, but as the code is written in direct style and we generate CPS code as output it is hard to see the relation between the written and generated code.

5.3.4 If we were to do it again would we take the same approach?

The answer is conditional, because of how complex the generator became because of the approach we took it was hard to implement all the generation strategies we came up with. We concluded that a lot of the generation strategies come down to rewriting pieces of the code, so they have the same semantics but are easier to generate. An example of this would be how we use parameters to functions (see section 3.2.8). When parameters to functions are function calls, we have troubles generating a CPS variant of this statement. We need to execute the functions in a specific order, assign the values the functions produce to a variable, and pass this variable as an argument to the function. The code in the next listing illustrates this.

```
1 //Hard to generate in cps, as we have to see functionB needs to be executed before functionA
2 functionA(functionB());
3
4 //Same semantics, easy generation, as we do not need to worry about the context in which the function
      call takes place
5 var tmp = functionB();
6 functionA(tmp)
```

Listing 5.2: Example of how generation would be easier with different semantics

The approach we took is a viable solution, and we can make a fully functional CPS compiler with it. If we were to do it again however, we would look into the possibility of creating an intermediate model between the AST and the generator. This extra model would be a transformed AST that would be semantically equivalent to the AST but where the elements would be in a form for which it is easier to generate CPS code, as shown in listing 5.2.

We would have to see if initial costs of creating a transformer that would create the intermediate model, would make creating generator so much easier to outweigh the costs. There are a lot these cases described in chapter 3 where an

intermediate model would possibly make creation easier. Therefore, we would at least look in to this possibility, if it is a better solution that the one we took by doing all the changes in the generator, we can not say.

Therefore the answer to the question "If we were to do it again would we take the same approach?" is conditional, it is dependant on the result of the research findings of the approach with the intermediate model.

5.4 Conclusion

As we have shown it is possible to create a CPS version of Co-op using Scala with closures. Full backwards compatibility is possible to achieve, as the syntax stays the same, and we are able to call Java functions in direct style that can be used in Co-op programs, in the new CPS version. The generation is not trivial as a lot of the elements of the AST of Co-op programs are context dependant in the CPS version, where they were not in the original Co-op version.

All in all we are satisfied with the choices we have made during the development. It turns out that using Scala with closures is a good option. We are also satisfied with how we implemented the function invocation. On how we implemented the generator we are not completely satisfied. We would like to have a fully functional CPS version of Co-op, but we do not. What we do have is generation strategies for the elements but not implemented them in the generator yet, because the generator is to complex to easily implement all the generation strategies in. Using the different approach for changing the generator described in section 5.3.4 this might not have happened. Therefore, it is unfortunate that we did not consider this, but chose to only change the generator. All in all our solution described in this thesis is a viable solution to change the generation of the Co-op language, from direct style, to CPS.

Bibliography

[1] M. Laarakkers, "Requirements for tailorable and composable control flow," Jul. 2012.

[2] L. M. J. Bergmans, S. te Brinke, C. M. Bockisch, and M. Akşit, "Free composition instead of language dictatorship," in *Proceedings of the 7th International Conference on Software Paradigm Trends, ICSOFT 2012, Rome, Italy.* SciTePress, July 2012, pp. 388–393.

[3] (2012, Nov.) The co-op project page @ONLINE. [Online]. Available: http://sourceforge.net/projects/co-op/

[4] H. G. GÃIJRBÃIJZ, "Programming language development," 2011.

[5] Java language specification. [Online]. Available: http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.20.3

[6] E. Lippert, "Continuation passing style revisited," 2010, all articles listed. [Online]. Available: http://blogs.msdn.com/b/ericlippert/archive/tags/continuation+passing+style/

[7] (2012) Continuation-passing style. [Online]. Available: Continuation-passingstyle

[8] Xtext 2.4 documentation. [Online]. Available: http://www.eclipse.org/Xtext/documentation/2.4.0/Documentation.pdf

[9] [Online]. Available: http://en.wikipedia.org/wiki/Function_object

[10] M. Odersky, P. ippe Altherr, V. Cremet, I. D. G. D. o chet, B. Emir, S. McDirmid, S. tÃl'phane Micheloud, N. Mihaylov, M. Schinz, E. S. man, L. S. o on, and M. Zenger. (2012, Nov.) An overview of the scala progra mming language @ONLINE. [Online]. Available: http://www.scala-lang.org/docu/files/ScalaOverview.pdf

[11] R. Hieb and R. K. Dybvig, "Continuations and concurrency," in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, ser. PPOPP '90. New York, NY, USA: ACM, 1990, pp. 128–136. [Online]. Available: http://doi.acm.org/10.1145/99163.99178

[12] [Online]. Available: http://en.wikipedia.org/wiki/Closure_(computer_science)

[13] "C# language specification," chapter 15. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa664602(v=vs.71).aspx

[14] [Online]. Available: http://en.wikipedia.org/wiki/Tail_call