Department of Computer Science

Master's Thesis

ScalaFlow

Continuation Based DataFlow Concurrency in Scala

Submitted on:

September 2nd, 2010

Submitted by:

B. Sc. Kai H. Meder
Im Dauenkamp 3
33332 Gütersloh, Germany
Phone: +49 (160) 80 55 0 77
E-mail: kai@meder.info

Supervised by:

Prof. Dr. Ulrich Hoffmann
Fachhochschule Wedel
Feldstraße 143
22880 Wedel, Germany
Phone: +49 (41 03) 80 48-41
E-mail: uh@fh-wedel.de

ScalaFlow

Continuation Based DataFlow Concurrency in Scala

Master's Thesis by Kai H. Meder

One of the biggest challenges in software development is leveraging the increasing number of physical and virtual cores. Reactive architectures promise improved scalability, but impose an inversion of control which further increases the complexity of parallel software. Out of the multitude of concepts and patterns, this thesis focuses on the DataFlow Model as a basis for concurrent programming. In this model control flow is determined primarily by the data itself, allowing the creation of programs with an uncluttered logical flow. This thesis builds on the hybrid object oriented and functional programming language Scala, and in particular its advanced feature of automatic transformation into Continuation-Passing-Style. It describes the design and implementation of the ScalaFlow framework, which allows concurrent DataFlow environments to be embedded into Scala projects.

Layout based on Björn Peemöller's, Stefan Roggensack's and Timo B. Hübel's template as well as LaTeX 2ε, KOMA-Script and BibTeX.

Contents

List of Listings

1

Introduction

Moore proposed his famous law in 1965 [Moo65], stating that the number of transistors on an integrated circuit would double roughly every two years. This prediction has been proven correct, allowing engineers to increase the performance of processors by a comparable factor. Recently though, the raw clock speed of processors has no longer been increasing at the same pace. Instead, the number of execution cores in each processor is increasing. This change lead to the coining of the popular term ”Multicore Crisis” [War07, Far08]. Developers must invest considerable effort in migrating software from a sequential design to a concurrent design, leveraging multiple physical and virtual cores. This is a complex task; there are multiple concurrency models and patterns, each with varying efficiency and implementation difficulty, depending on the specific problem to be solved.

1.1. Motivation

Creating concurrent programs using the Java programming language relies on threads and monitors, within a model of shared mutable state [GBB+06]. Thread-safety is not

easy to accomplish; excessive use of concurrency primitives easily leads to deadlocks [Rep10a], while insufficient synchronization results in data-races [Rep10b]. One major shortcoming of threads is that although lightweight compared to processes, their number is limited to some thousands on a common JVM and memory configuration. This limit impedes out-of-the-box scalability and can be a major bottleneck when creating networking programs or distributed systems. Worst affected are programs which serve a large amount of concurrent connections, e.g. web servers with numerous client connections. The computation handling each connection may suspend and resume its underlying thread at any time, depending on availability of data at the associated TCP socket.

The general solution to scale the number of possible concurrent connections beyond the JVM imposed thread-limit is to employ the *Reactor Pattern* [Sch94] to process asynchronous I/O. However this imposes inversion of control [Fow05] on the program, increasing the semantic distance between the programmer's intention and the actual code.

Scala [Ode10], a statically typed programming-language which directly compiles to Java bytecode (and MSIL as well), is able to use the Java infrastructure, including Java's concurrency constructs. However Scala's most prominent feature for concurrency oriented programming is the *Actor Model*, adapted from Erlang [HO07, AVWW96]. The model provides stateful objects (the actors) which communicate solely by passing immutable Messages. Every actor holds an exclusive mailbox (queue), which stores incoming messages in 'first-in, first-out' (FIFO) order. Messages are sent asynchronously, often described as "fire and forget", as there is neither a wait for reception nor guaranteed delivery. This model scales to millions of different actors on a single JVM. Since the virtual machine processes actors using a thread pool, each actor is required not to block its underlying thread, which effectively rules out traditional blocking I/O. There is no further experience whether the actor model is suited for processing asynchronous I/O. The control flow of an actor-based program is determined by the communication via messages, with the program logic fragmented among the participating actors.

The goal of this work is to support the development of scalable concurrent programs without imposing inversion of control. Therefore this thesis defines and focuses on the *DataFlow Model*, which is based on the declarative *Data-Driven Concurrency Model*. The foundations of the DataFlow Model are single-assignment variables and communication channels. The control flow is suspended when reading an unassigned

variable and resumed on assignment. Similarly, consuming an empty channel suspends and filling the channel resumes suspended consumers. Channels provide flow-control modes that transparently suspend and resume producers as well. This allows one to write concurrent programs with uncluttered logical flow, which transparently suspend or resume their execution based on the availability of data.

Scala is the language of choice to implement the DataFlow Model as it combines object-oriented programming with functional programming, offers a strong type system and supports *Delimited Continuations* [RMO09]. Continuations can be captured as first-class values represented by functions. By capturing continuations, computations can be suspended without blocking threads. Thus, continuations enable programs to scale beyond any thread-limit by providing a thread-independent mechanism to transparently suspend computations.

1.2. Scope

This thesis describes the design and implementation of a framework supporting creation of concurrent programs in Scala. The *ScalaFlow* framework provides a terse, internal *Domain Specific Language* [OSV08, Eva04] (DSL) to define concurrent computations, based on the *DataFlow Model*. Building on the basic ScalaFlow components, an I/O library is provided to create networking applications and distributed systems.

The framework's basic design is based on *Flow*s which represent and initialize concurrent computations. Flows can communicate with the main program by returning a *Future* [Rep10c]. *Variable*s are the basic building blocks in a DataFlow environment. They can be bound to a value only once and suspend their readers until bound. *Signal*s are based on suspending *Variables*. They do not contain a value but allow code to wait for the signal's invocation by suspending. *Channel*s are bounded sequences of variables that can be shared used for communication between concurrent computations. Channels have queue semantics; multiple concurrent readers compete against each other to consume the channel's head element. Consumers are transparently suspended when they attempt to read from an empty channel. Channels provide lazy, eager and bounded buffer flow-control to suspend producers based on their capacity. Additionally, channels provide higher-order functions like `foreach`, `map`, `filter`, `fold` and several others. *Pipeline*s combine multiple channels

into consecutive stages of processing. Each stage is defined by a function which concurrently and continuously computes the transformation from its input channel to its output channel. The final basic component of the framework is the I/O library. *Socket*s represent network connections whose readers and writers are suspended and resumed based on the availability of data. *Acceptor*s listen on server sockets and accept new client connections (creating a new Socket for each connection accepted). Both sockets and acceptors are based on a lightweight event dispatcher for Java's NIO library. Furthermore the framework features a HTTP tokenizer and parser to process the `HTTP` specification. An example demonstrating a basic web server is listed in chapter 6.

1.3. Related Work

This thesis was inspired by Jonas Boner's dataflow implementation [Bon09] which is based on Scala Actors and blocking Queues.

The *Scala.React* library [Mai10] is currently under development in the scala-incubator. The library provides reactive combinators [Ell09] based on a synchronous dataflow [BCLGH93] core to "deprecate the observer pattern" [MRO10, GHJV95].

1.4. Outline

Prior to the development of the framework, the key concepts of the DataFlow model are outlined in chapter 2. Selected features of the Scala programming language follow in chapter 3, including the core feature of this thesis: delimited transformation into Continuation-Passing style.

The development of ScalaFlow is described in chapters 4 to 5. In chapter 4 the overall design of the ScalaFlow framework is developed. Starting with an applicable parallel programming model, Java's and Scala's concurrency constructs are presented and the scheduling is discussed. The key concept of suspending and resuming computations via CPS-Transformation is explained, along with the motivation for this design choice. Building on this, the basic design of the framework components follows. The individual components are refined incrementally, leading towards the final design.

Chapter 5 presents ScalaFlow's networking capabilities. The motivation for using asynchronous I/O is detailed and a lightweight NIO dispatcher is developed. This leads into the detailed design of the network components Socket and Acceptor. The chapter ends by extending ScalaFlow's I/O subsystem with general HTTP processing capabilities.

Based on the functionality depicted in the previous chapters, chapter 6 presents several use cases and sample applications. The first set of use cases demonstrate the basic intended usage of the framework. The second set demonstrate networking, followed by more elaborate examples such as a web server.

A note regarding listings: all ScalaFlow code excerpts are shortened to a minimum to focus on key features. All visibility modifiers like `private` and `protected` are omitted for brevity. If irrelevant, variables, methods and functions are only denoted by a type and not bound to any value or implemented. Lines prefixed with "scala>" denote input to the Scala Interpreter shell.

2

DataFlow Model

This chapter introduces the declarative *Data-Driven Concurrency Model*[1] and extends it with nondeterminism to create the *DataFlow Model*, specific to this thesis. The first section 2.1 introduces the basic declarative model of programming and then extends it with concurrency and dataflow variables to create the *Data-Driven Concurrency Model*. The shortcomings of the deterministic model are discussed, and extensions introduced to address them in section 2.2. The last section 2.3 defines the *DataFlow Model*, combining *Data-Driven Concurrency Model* with non-declarative constructs, as specified by a *Domain Specific Language* (DSL) within Scala.

2.1. Data-Driven Concurrency Model

The *Declarative Model* is based upon the principle that a program's output is a mathematical function of its input. A declarative operation is independent of outside state, stateless and deterministic. Functional and logical programming languages

[1] This chapter is based on the book *Concepts, Techniques, and Models of Computer Programming* [RH04]

like *Haskell* (mathematical functions) and *Prolog* (logical relations) are declarative programming languages. Declarative programming as a specification can *formally* make the implementation dispensable, since the specification *is* the program. Practically however, declarative programming still requires an *efficient* implementation. Thus a means of transforming specifications into an efficient executable programs is required, either automatically or manually.

The declarative model is based on a single-assignment store, which is a set of initially unbound variables. Once a variable is bound, it stays bound throughout the computation and is indistinguishable from its value. In contrast to functional programming languages, the creation of unbound variables and their binding to a value are separate steps. However the binding is monotonic; the binding adds information that can not be changed or undone afterwards. Thus single-assignment variables are still declarative rather than imperative.

Variables that cause the program to wait until they are bound are called *dataflow variables*. It is unreasonable to use such variables in a sequential program since reading an unbound variable would cause the program to wait forever. In a concurrent program however, where multiple computations are executed at the same time, dataflow variables can be used to great effect. The meaning of "at the same time" depends on the environment: on a single processor machine the order of concurrent statements is interleaved (by a nondeterministic scheduler). By contrast on a multi processor machine, statements from several computations are executed simultaneously (in parallel). As there is always at least one interleaving that is observationally equivalent to the parallel execution, the parallel execution can be described in terms of the interleaved sequence [RH04].

The *Data-Driven Concurrent Model* extends the declarative model with concurrency but remains declarative. Therefore the model provides *declarative concurrency*. A concurrent declarative program does not need to terminate, instead it computes incrementally since its input can grow indefinitely: the input might contain unbound variables, therefore the input is extended when binding the variables. It is declarative in the sense that the following axiom holds for all possible inputs:

- All executions do not terminate,
- Or all executions reach partial termination and produce logically equivalent results.

  - Where partial termination means that the program will eventually stop executing if its input stops growing.
  - Where two results R_1 and R_2 are logically equivalent if they contain the same variables and for each variable V_i, `values(`V_i`,`R_1`) = values(`V_i`,`R_2`)` (they define the same set of values).

The data-driven concurrent model is declarative and therefore by definition deterministic, despite the fact that the interleaving of its computations is nondeterministic. By using dataflow variables, the *nondeterminism is effectively hidden* though:

- Dataflow variables can only be bound once, thus scheduler nondeterminism only affects the exact moment of binding.
- Any operation reading a variable has no choice but to wait until the variable is bound, therefore the nondeterminism does not become visible.

Any deterministic model is also inherently *race free*, since race conditions result from observable nondeterministic behaviour. Dataflow variables are order-independent and replace static dependencies with *dynamic dependencies* that are determined by the data itself. The output of one computation can be passed as the input to another computation, independent of the order in which the two computations are executed. Furthermore, the reader of a dataflow variable *implicitly synchronizes* on the availability of the variable's value, so no synchronization operations are directly visible.

A dataflow variable can be considered as a communication channel. Binding the variable is sending, reading a receive operation. The semantics of the channel are that only one message can be sent, but this message can be received many times. Sending is an asynchronous operation whereas receiving is synchronous. It is possible to create synchronous as well as asynchronous operations:

- *Asynchronous* receiving can be achieved by using the variable without reading it.
- If by-need triggers are supported, a *synchronous* send can be achieved by using an additional variable V_2 that pauses the sender until it's bound: the sender registers a by-need trigger that binds both the channel variable V_1 to the sent value and the additional variable V_2 to a value that continues the sender's execution.

However, it is not possible to define a non-blocking send operation which instantly returns whether sending succeeded. In order to determine success, the variable had to be checked whether it is already bound, which is an invalid nondeterministic choice in the declarative model. Summarized, a communication channel based on dataflow variables is able to support both asynchronous and synchronous operations.

Based on dataflow variables, it is possible to define *streams* that are composed of a potentially unbounded list of dataflow variables, whose last element is an unbound dataflow variable. Naturally, they can also be considered as a communication channel. Receiving is done by reading the first element, sending by extending the stream with one element. This communication channel allows multiple messages to be sent by extending the stream. Since the last element is always an unbound dataflow variable, execution of the receiver is paused when it attempts to read an empty stream.

2.2. Introducing Nondeterminism

Due to its deterministic nature, a declarative model is fundamentally limited in its expressiveness, because independent concurrent computations behave nondeterministically in respect to each other. Using the analogy of a client/server application, the order in which messages are received by the server from multiple independent clients is nondeterministic. Moreover, it is not possible to share one communication channel between multiple clients, since the clients' bindings to the same channel would conflict with each other. Using dedicated channels per client is not possible either, since the server would have to make a nondeterministic choice which channel to read next and which channel to skip, because it is empty.

Furthermore, it is not possible to handle failure without giving up deterministic behaviour. An exception (or any other signal of failure) is raised either due to nondeterministic execution or due to an operation that was attempted without its preconditions being met. As specified, the order of execution of concurrent computations is nondeterministic, which does not cause a nondeterministic result. However, if multiple computations bind the same variable concurrently, only the first (nondeterministically chosen) computation will succeed and determine the overall result while all other computations will fail. Likewise, if an operation fails and raises an exception, the operation's effect is not specified and the result has potentially become nondeterministic.

One of the main reasons for the introduction of nondeterminism is to abstract from details of implementation. There may be many different implementations of a computation, each with an observably different pattern of behaviour. Nondeterminism arises from a deliberate decision to ignore factors which influence execution order in concurrent programming. It is useful for maintaining a high level of abstraction in descriptions of the behaviour of both physical and data processing systems [Hoa85].

To summarize the preceeding sections, dataflow variables can be used to great effect in concurrent programs as they provide declarative single-assignment, race-free access, implicit synchronization and permit data-dependent dynamic dependencies between computations. Dataflow variables are in the borderland between stateless and stateful programming. On the one hand, a dataflow variable is stateful because it changes state from unbound to being bound. On the other hand, it is stateless because binding is monotonic (see section 2.1). The stateful aspect is required for communication between concurrent computations and the stateless aspect can be used to gain the advantages of declarative programming within a more expressive nondeclarative model. Therefore dataflow variables are a reasonable tool to create concurrent components with dataflow execution, even within a nondeclarative language such as Scala.

2.3. DataFlow Semantics

This section specifies a *Domain Specific Language* within Scala, implementing the *DataFlow Model*. This model provides dataflow execution and combines dataflow variables from the *Data-Driven Concurrency* model with nondeterministic communication channels.

The most basic operation is `flow` that schedules a computation for concurrent execution (see listing 2.1). Scheduling of multiple computations is nondeterministic, but the instructions comprising each computation are executed in sequential order. By applying `flow` to a computation, the computation becomes a dataflow execution in which dataflow variables and channels can be used. Using the operations outside of a flow yields unspecified behaviour. `val` denotes an immutable value, which may be a dataflow variable.

```
flow {
  val x = 30 // value, not a dataflow variable
  val y = 12
  x + y
}
```

Listing 2.1: Creating a concurrent computation

Applying `flow` to a computation returns a *Future* that represents the computation's result. The future can be applied to get its result (see listing 2.2). When the computation has not yet produced a result, the caller is blocked until the result is available (i.e. the computation has returned). The Future is the only point of data exchange between the dataflow execution and its enclosing code. In summary, a flow transforms a computation into dataflow execution and returns a future used to get the computation's result after it has been completed.

```
val fut = flow { 30 + 12 }
val res = fut()
```

Listing 2.2: Flow Result

Unbound dataflow variables can be created by instantiating the `Variable` class (see listing 2.3). Any computation that attempts to read an unbound variable is suspended. The computation is resumed when the variable is bound to a value. Variables can be created outside the dataflow environment, reading or binding is not possible though.

```
// immutable value containing a dataflow variable
val v = new Variable[Int]

flow { v() + 2 } // reading
flow { v := 1  } // binding
```

Listing 2.3: Variable

Communication channels that can be shared among multiple flows are created by instantiating the `Channel` class. A channel provides similar suspend/resume behaviour to a `Variable`: if empty, readers will be suspended. When an empty channel is filled with new elements, suspended readers are resumed in FIFO order, thus implementing the semantics of a *Queue*.

```
var c = new Channel[Int]
flow { // dequeing
```

```
  c() + c()
}
flow { // enqueing
  c << 1
  c << 2
}
```

Listing 2.4: Channel

A channel provides flow-control: producers may be suspended, based on the channel's capacity. A channel created with an capacity of zero operates in lazy mode: producers are suspended and resumed based on consumer demand. If the channel has an infinite capacity (approximated by Int.MaxValue), the channel is in eager mode: producers are never suspended and can produce values independent of the consumers (in practice, producers are suspended when the channel's capacity of Int.MaxValue is reached). Any value in between defines the channel as a bounded buffer: the producers can produce values until the channel is filled up to its capacity. When consumers drain the channel, such that the number of elements is below its capacity, the producers are resumed in FIFO order.

To iterate over a channel's values while maintaining the suspend/resume semantics, the `Channel` provides two higher-order functions (see listing 2.5): `foreach` takes a function of type `A => U` that is successively applied to all elements of a `Channel[A]` and returns an arbitrary type `U` (the returned value is discarded, `U` is a hint at the `Unit` type). Invoking `foreach` suspends the caller until `foreach` returns control. Control is returned when `foreach` has successively applied the passed function to each channel element.

The second higher-order function `map` is executed concurrently and does not suspend the caller. `map` takes a function of type `A => B` and returns a newly created `Channel[B]`. The passed function is successively applied to each channel element. The return values are put into the new `Channel[B]`. In summary, invoking `map` concurrently executes the passed function and immediately returns a new channel that is filled by the output of that function when applied to the values in the passed channel.

```
var xs = new Channel[Int]
flow {
  // suspends while executing
  xs.foreach(x => println(x))
}
flow {
```

```
  // concurrently executing
  val ys = xs.map(x => 10*x)

  // suspends while executing
  ys.foreach(y => println(y))
}
flow { xs << 1; xs << 2; xs << 3;  }
```

Listing 2.5: Channel with competing applications of `foreach` and `map`

Applications of `foreach` and `map` can be abbreviated using a `for`-comprehension. Listing 2.6 demonstrates using both `foreach` and `map` with the `for`-comprehension. The application of `map` is denoted by the `yield` keyword.

```
var xs = new Channel[Int]
flow {
  // map, concurrently
  val ys = for (x <- xs) yield
    10*x

  // foreach, suspending
  for (y <- ys)
    println(y)
}
flow { xs << 1; xs << 2; xs << 3;  }
```

Listing 2.6: Channel with For-Comprehensions using both `foreach` and `map`

Until now, application of `foreach` will never return control to the caller, since it will encounter the empty channel when all elements have been processed. This suspends `foreach` indefinitely, resuming when another element is put into the channel. The same applies for `map`, with the distinction that `map` is concurrently executing and thus its suspension does not affect the caller. To signal an End-of-Stream to the channel, the channel can be terminated (see listing 2.7). Consumers of an empty and terminated channel are resumed by raising an `TerminatedChannel` Exception. This requires consumers to be termination-aware to catch this exception appropriately. `foreach` and `map` silently catch the exception and return control to the caller. Therefore, `foreach` will return control to the caller if a terminated channel has been fully processed, and is now permanently empty. Terminating an already terminated channel or putting elements to an already terminated channel will also raise a `TerminatedChannel` Exception.

```
var xs = new Channel[Int]
flow { for (x <- xs) println(x) }
```

```
flow {
  xs << 1; xs << 2; xs << 3;
  xs <<#; // terminate channel
}
flow { xs() } // competing with first flow.
              // depending on order of execution,
              // this might raise an exception if
              // scheduled too late, when the
              // terminated channel is already empty!
```

Listing 2.7: Channel Termination with For-Loop

In conclusion, the *Data-Driven Concurrency Model* has been extended with nondeterminism to allow for nondeterministic communication channels. This model is referred to as the *DataFlow Model* throughout this thesis and has been specified by a *Domain Specific Language* within Scala. The DSL provides constructs to create and process dataflow variables and channels and operations to create concurrent computations using dataflow execution.

3

Aspects of Scala

The ScalaFlow framework presented in this thesis is implemented using the Scala 2.8 release in combination with the *Delimited Continuations* compiler plug-in. Scala is a hybrid programming language, fusing object-oriented and functional programming. As the reader is expected to be familiar with these basic principles, Scala's most basic syntax and features will not be described here, but are available at [O$^+$].

Section 3.1 describes the object-oriented features of Scala, followed by section 3.2 covering the functional features. Scala is statically typed and features a very powerful type system, including *Type Inference* and *Higher Kinded Types*. As Scala claims to be a 'scalable' language, prominent features such as for-comprehensions are 'syntactic sugar' for arbitrary control structures using a monadic style. The transformation of for-loops is described in section 3.3. Finally section 3.4 introduces the concept of Delimited Continuations via a type-directed, selective transformation into Continuation-Passing Style. This feature is heavily used in this thesis to suspend and resume computations.

3.1. Object Orientation

Scala is based on Java's core capabilities as it compiles natively into Java bytecode. Therefore the same basic principles and features of object oriented programming apply to Scala. Scala also inherits Java's basic shortcomings, such as type erasure [Comb]. The foundation of the type system is a class hierarchy, starting with the `Any` type which has `AnyVal` and `AnyRef` as subtypes. Java's reference types and the basic `ScalaObject` are descendants of `AnyRef`. There are two bottom types: `null` is a subtype of `AnyRef` and therefore is not assignable to `AnyVal`s, while `Nothing` is a subtype of every type. However, there is no value of type `Nothing` as this type signals abnormal termination. Value classes (`Byte`, `Int`, `Double`, `Boolean`, etc.) are subtypes of `AnyVal`; they are mapped to Java's primitive types. Finally, there is a special value class `Unit` which corresponds to Java's `void` type. Its single instance value is `()`.

An object is an instance of a class. Classes contain fields and methods. *Companion Objects* are the global singleton instances of their associated classes and replace Java's static members. They share namespace and scope with their class, so a companion object can access private fields and methods of its defining class (unless prohibited by the `private[this]` modifier).

Scala differentiates (by paradigm) between mutable and immutable classes and fields. Instances of mutable classes can alter their internal state, whereas the state of immutable objects can not be modified after construction. Therefore operations on immutable classes return newly constructed objects to represent the modified state. Scala's collection library offers mutable as well as immutable implementations. Methods may be overloaded as in Java, the resolution is done on the static types of the arguments.

In contrast to Java, there are no raw or primitive types in Scala. Every type is a class and all operations are methods on objects, e.g. `1+2` is translated to `(1).+(2)`. Every method, regardless of its number of arguments, may be written in infix-operator style. (e.g. `str.indexOf('a', 13)` may be written as `str indexOf ('a', 13)`). Additionally, Scala features prefix and postfix operator notation for unary methods. Names of methods or operator identifiers may consist of most of the printable `ASCII`-characters, including `+, :, ?,  and #`.

Instead of interfaces, Scala features *Traits*, which are the fundamental unit of code reuse. Traits mix the concepts of interfaces and abstract classes: they specify interface-like contracts by defining methods and fields. But unlike Java's interfaces, they also contain pre-implemented code similar to an abstract class. A class is allowed to inherit ("mix in") multiple traits. Traits can augment classes by adding new methods. This allows to create thin interfaces and then extend them to rich interfaces by mixing in traits containing the auxiliary operations. Apart from adding new behaviour, traits can modify behaviour by overwriting methods. A class may mix-in multiple traits which all modify one specific method. The resulting class contains all these modifications by combining them in a specific order, so traits can function as "stackable modifications" (see listing 3.1). The order in which the traits are mixed-in is significant and the actual linear order of super-calls within the traits is determined by a linearization-process [Ode10, p.56]. This stackable and linearized order of super-calls differentiates traits from multiple inheritance.

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int): Unit
}
class BasicIntQueue extends IntQueue {
  import scala.collection.mutable.ArrayBuffer
  private val buf = new ArrayBuffer[Int]
  def get(): Int  = buf remove 0
  def put(x: Int) = buf += x
}
trait Inc extends IntQueue {
  abstract override def put(x: Int) = super.put(x + 1)
}
trait Double extends IntQueue {
  abstract override def put(x: Int) = super.put(x * 2)
}

scala> val q = new BasicIntQueue with Inc
q: BasicIntQueue with Inc = $anon$1@113e371
scala> q put 41
scala> q get
res5: Int = 42

scala> val q2 = new BasicIntQueue with Inc with Double
q2: BasicIntQueue with Inc with Double = $anon$1@c9e870
scala> q2 put 41
scala> q2 get
res7: Int = 83
```

Listing 3.1: Demonstrating traits as stackable modifications

3.2. Functional

Scala is functional according to the definition of having functions as first-class values. Functions can be written as unnamed literals (see listing 3.2) and passed as values.

```
scala> val increase = (x:Int) => x + 1
increase: (Int) => Int = <function>
scala> increase(41)
res1: Int = 42
```

Listing 3.2: Defining a successor function by a function-literal

Scala features currying and partially applied functions. As functions are first-class value and therefore can be passed as parameters to other functions, it is possible to define higher-order functions: functions applied to functions (see listing 3.3).

```
scala> def modify(f: Int => Int)(x: Int): Int = f(x)
modify: ((Int) => Int)(Int)Int
scala> val m = modify _ // partially applied
scala> val f = m(incFun) _ // curried application
f: (Int) => Int = <function>
scala> f(41)
res1: Int = 42
```

Listing 3.3: Higher-order curried function

It is possible to define closures 3.4. Closures are functions containing values passed as arguments (bound variables) and additional free variables. The values of the free variables are captured in the surrounding scope of the closure definition.

```
scala> val x = 11
scala> val fun = (i:Int) => i + x
closure: (Int) => Int = <function>
scala> fun(31)
res1: Int = 42
```

Listing 3.4: Closure capturing the value-binding of its free variable x

3.3. For Comprehension

Scala differentiates between *for loops* and *for comprehensions* [Ode10, p. 89]. For-loops of type `for (enums) expr` execute the expression `expr` for each binding generated by the enumerators `enums`. In contrast, for-comprehensions of type `for`

`(enums) yield expr` execute `expr` for each binding generated by `enums` and collect the results.

The for construct is syntactic sugar for the applications of higher-order functions. For-loops are expanded into applications of `foreach(f: A => Unit): Unit` whereas for-comprehensions expand to applications of `map(f: A => B): C[B]` and `flatMap(f: A => C[B]): C[B])`, respectively. Additionally, it is possible to define *guards* of type `for (enums if guard) ...` which expand to applications of `filter(f: A => Boolean)`. The expansion does not constrain the specific types of the functions. Therefore, `foreach` can be defined as `foreach(f: A => U): U` if this definition satisfies the type-checker.

The expansion allows to write libraries that can be used in for-loops and for-comprehension by providing the higher-order functions. Furthermore, arbitrary control-structures can be used by for-comprehensions, they resemble Haskell's do-notation that is used for monadic applications [Coma].

The expansion of the for-loop is demonstrated in listing 3.5, whereas the for-comprehension is listed in 3.6 and 3.7. The expansion of guards is demonstrated in listing 3.8.

```
val xs: List[Int]
for (x <- xs) { println(x) }
// expands to:
xs.foreach( x => println(x) )
```

Listing 3.5: For-loop expanded to an application of `foreach`

```
val xs: List[Int]
val ys = for (x <- xs) yield { 2*x }
// expands to:
val ys = xs.map( x => 2*x )
```

Listing 3.6: For-comprehension expanded to an application of `map`

```
val xs: List[List[Int]]
val ys = for (ys <- xs; y <- ys) yield { 2*y }
// expands to:
val ys = xs.flatMap(ys => ys.map(y => 2*y))
```

Listing 3.7: For-comprehension expanded to an application of `flatMap`

```
val xs: List[Int]
val ys = for (x <- xs if x > 3) yield 2*
// expands to:
val ys = xs.filter(x => x > 3).map(x => 2*x)
```

Listing 3.8: Guarded for-comprehension expanded to applications of `map` and `filter`

3.4. Delimited Continuations

In general, a continuation contains the remaining instructions of a computation, derived from the point where the continuation is captured. Continuations are reifications of computations, therefore undelimited continuations embody the whole execution state including the full stack. Capturing continuations originates in scheme's `call/cc` control operator [Comc].

In contrast, *Delimited Continuations* only encompass a specific part of a computation and as such only embody a partial execution state. The scope or the delimitation of the delimited continuation is explicitly specified in the code. After invocation, delimited continuations return control to the caller and are able to return values as well. It is possible to execute delimited continuations multiple times with the main process continuing at the call-site after invocation.

There are several techniques for capturing continuations. One method is to save the stack and restore it upon reification of the continuation. The other method, used by Scala's *Delimited Continuations* compiler plug-in [RMO09], is to selectively transform the program into *Continuation-Passing-Style* (CPS). A program written in imperative style applies functions that return values to the caller. The computation then continues by invoking other statements and waits for their result. In contrast, a program written in CPS calls a function and additionally passes the continuation of the program, which is a first-class value and represented by a function. The function never returns to the caller, instead the function can apply, transform or even discard the passed continuation to continue the computation. Therefore the control flow is explicitly specified by functions that always pass the computation's continuation. As a consequence, the state of the program is always stored in closures bypassing the stack.

Instead of transforming every function into CPS, Scala performs a selective transformation which only transforms functions that are required to pass along continuations. The transformation is type-directed by annotations of the form `@cps[A]` (actually by fully qualified annotations of the form `@cpsParam[A,B]`). The annotation `@cps[A]` denotes that its function's continuation returns type `A`, whereas the annotation `@suspendable` is a type-alias for `@cps[Unit]` and denotes functions which are meant to only suspend and whose continuations return `Unit`. As the transformation is type-directed only functions providing the cps-annotations are transformed. Non-transformed functions still use the stack, which avoids problems from keeping the state only in closures on a stack-based architecture like the JVM.

The primitive operations in the CPS context are `reset` and `shift`. The operation `reset` delimits the range of the continuations, whereas `shift` captures the current continuation up to the next enclosing reset as a function. CPS-transformed code is executed in a continuation-monad of type `ControlContext[A,B,C]` where type `B` and `C` are denoted by the annotation `@cpsParam[A,B]`. Listings 3.9, 3.10 and 3.10 show the usage of `reset` in conjunction with `shift`.

```
val r = reset {
  shift { k: (Int=>Int) =>
    // not invoking k
  }
  // continuation starts here
}
r: Unit = ()
```

Listing 3.9: Capturing a continuation without invoking it

```
scala> val r = reset {
  val kArg = shift { k: (Int=>Int) =>
    k(42)
  }
  // continuation starts here, parameter = kArg
  kArg
}
r: Int = 42
```

Listing 3.10: Capturing the continuation and invoking it once with 42

```
scala> val r = reset {
  val kArg = shift { k: (Int=>Int) =>
    k(k(19))
  }
  // continuation starts here
```

```
  kArg + 1
} * 2
r: Int = 42
```

Listing 3.11: Capturing the continuation and invoking it twice

The `while` loop is rewritten by the CPS plugin to a recursive function application. This makes it possible to use loops in a CPS context. However a large loop iteration count will overflow the call stack. This problem can be solved by using trampolining [Dou09, Ode08] to emulate tail-call optimisation. Another workaround is to throw exceptions which Scala's Actor library uses to unwind the call stack [HO07]. Furthermore, it is not possible to use the for-loop on arbitrary collections: the collections' definitions of `foreach` and `map` are not applicable inside the CPS context because they lack the required `@cps[T]` annotation.

4

Design and Implementation

This chapter describes the implementation of the ScalaFlow framework and discusses the design decisions made. The ScalaFlow framework provides operations to create concurrent computations based on the DataFlow model. It is implemented using Scala 2.8 and uses Scala's transformation into Continuation-Passing-Style to suspend computations. The chapter starts by detailing why continuations were chosen as the mechanism for suspending computations, rather than Java monitors (in section 4.1. Section 4.2 discusses how computations enter the framework and how they are processed and scheduled. The foundation of the DataFlow Model is the single-assignment variable, which is designed and incrementally implemented in section 4.3. Communication channels introduce nondeterminism into the DataFlow Model. Their basic implementation is discussed in section 4.4.1. Subsequently channels are extended with flow control and a termination mechanism in section 4.4.2. Finally, section 4.4.3 splits the channel's implementation into different traits, defines DSL-aliases and implements multiple higher-order functions. The chapter ends by combining channels to form concurrent pipelines in section 4.5.

4.1. Monitors vs. Continuations

This section explains the decision to use continuations to suspend and resume computations, instead of using Java monitors.

Java provides *Wait & Notify* monitors that can be used to pause computations by suspending their thread. In the following, operations that suspend threads are called *blocking* operations to distinguish the behaviour from other suspending methods. Invoking `wait` on an object blocks the calling thread, while invoking `notifyAll` on an object unblocks all waiting threads. This can be used to design DataFlow variables: programs reading a variable first acquire the variable's monitor. If the variable is unbound, the accessor calls `wait` to block the computation's underlying thread. Computations that bind a variable to a value also acquire the variable's monitor, set the value and then call `notifyAll` to resume any blocked threads. This design occupies one thread per concurrent computation. JVM threads are a limited resource and context switches between threads are expensive. Therefore, this design has limited scalability.

This problem can be circumvented by using the `Task Pattern`, where each computation is represented by an independent task that is submitted to a managed pool of threads. The tasks are concurrently executed on the pool's threads. However if the pool is limited in its size, this can lead to a starvation problem: if all running tasks block their threads, no further task will be executed until a thread is resumed and freed. Submitting numerous tasks that read an unbound dataflow variable could block all of the pool's threads. The task binding the variable and thus resuming the waiting threads will not get executed: the blocked threads "starve". The starvation problem can be solved by using a dynamically adapting thread-pool. However this solution again raises the problem of threads as a limiting resource.

This thesis uses a different approach to suspend computations: Scala's *Delimited Continuations*. Each computation is transformed into Continuation-Passing-Style (CPS) which allows to capture the computation's continuation at arbitrary points of execution. DataFlow computations are executed inside CPS' `ControlContext` monad by applying the CPS primitive `reset` with the unevaluated computation. Invoking the second CPS primitive `shift` captures the computation's continuation. If the continuation is captured, the computation stops and its future statements are embodied in the continuation: the continuation is suspended. Executing the continuation resumes the computation. In the following, operations that suspend

a computation by capturing its continuations are called *suspending* operations to distinguish them from (thread-)*blocking* operations.

Since the DataFlow Model is a model of concurrent execution, Java monitors are still required to synchronize access to critical regions. However the monitor-based wait mechanism will not be used to suspend computations. To achieve concurrent execution, computations are executed simultaneously on multiple threads. A managed thread pool of fixed size will not suffer from starvation if the submitted tasks do not block their underlying thread. Since the tasks are *suspended* by capturing their continuations, the underlying threads will immediately be available to execute other tasks. A computation is resumed by submitting its continuation to the pool as a separate task.

Concluding this section, it is possible to suspend computations without blocking any thread by capturing their continuations. This allows concurrent execution of an unbounded number of computations using a fixed number of threads, without suffering from starvation.

4.2. Flows and Scheduling

This section discusses the way DataFlow computations are created. A solution to exchange data between concurrent DataFlow computations and the main non-CPS-transformed program is developed. Subsequently, the scheduling of DataFlow computations is discussed and multiple scheduler implementations are developed.

A DataFlow computation that enters the ScalaFlow framework is called a *flow*. Scala provides 'by-name' parameters that capture passed expressions without evaluating them (non-strict evaluation). This allows passing of unevaluated computations with a terse syntax and evaluation on demand, which is demonstrated in listing 4.1.

```
def flow(computation: => Unit): Unit = {
  print("captured: ")
  computation // execute
}
flow { println("hello world") }
```

Listing 4.1: Capturing unevaluated computations using a by-name parameter

To allow continuations to be captured, the computation needs to be applied to the CPS primitive `reset`. After this, the computation can be scheduled for execution as a suspendable DataFlow computation (see listing 4.2).

```
def flow(computation: => Unit @suspendable): Unit = {
  scheduler.execute( reset(computation) )
}
```

Listing 4.2: Applying reset before scheduling the by-name computaion

Flows return a *Future* to the main (non-CPS-transformed) program, which represents the flow's yet-to-be-computed result. To wait for this result the main program's thread has to be blocked. In contrast, flows that wait for other flow results must wait by suspending. The different 'await' actions are expressed by distinct traits that provide methods to `await` by *blocking* and to `sawait` by *suspending* (see listing 4.3). The methods return a `Boolean` to signal if the await was interruped (denoted by `false`). Moreover, it allows joint awaits using a logical AND, e.g. `blockingAwaitA.await && blockingAwaitB.await`.

```
trait BlockingAwait {
  def await: Boolean
}
trait SuspendingAwait {
  def sawait: Boolean @suspendable
}
```

Listing 4.3: Interface to await completion

The future result returned by flows is implemented by the class `FlowResult[A]` where type `A` denotes the flow's resulting value. It implements both the `BlockingAwait` trait and `SuspendingAwait` trait to await the flow's completion. Likewise, it provides the methods `get` to obtain the future value by blocking and `sget` to get the value by suspending until the flow has completed. This is demonstrated in listing 4.4 and 4.5.

```
def flow(computation: => A @suspendable): FlowResult[A] = {
  val result = new FlowResult[A]
  scheduler.execute(reset {
    result.set( computation )
  })
}
```

Listing 4.4: Flow returning a `FlowResult`

```
class FlowResult[A] extends BlockingAwait with SuspendingAwait
   {
  def set(v: A): Unit

  def await: Boolean
  def sawait: Boolean @suspendable

  def get: A
  def sget: A @suspendable
}
```

Listing 4.5: Interface of `FlowResult`

The implementation of the `FlowResult` uses a `CountDownLatch` (package `java.util.concurrent`) to implement blocking behaviour for the `await` and `get` methods. The latch is initialized with a count of one. By invoking the `FlowResult`'s `set` method, the latch is counted down to zero which unblocks threads that are `await`ing the latch. The suspending `await` and `get` methods use a dataflow variable, that is yet to be explained. Variable will provide the methods `set` and `get`, where `get` suspends the caller if the variable is unbound (see listing 4.6).

```
class FlowResult[A] extends BlockingAwait with SuspendingAwait
   {
  val latch = new CountDownLatch(1)
  val dfvar = new Variable[A]

  def set(v: A): Unit = {
    dfvar.set(v)
    latch.countDown
  }
  def await: Boolean = {
    latch.await
    true
  }
  def sawait: Boolean @suspendable = {
    dfvar.get
    true
  }
  def get: A = {
    latch.await
    dfvar.NonSuspendingDirectAccess
  }
  def sget: A @suspendable =
    dfvar.get
}
```

Listing 4.6: Implementation of `FlowResult`

To conclude the development so far, flows enter the ScalaFlow framework as a by-name computation. The computation is applied to `reset` to enable delimited continuations and then scheduled for execution. After the computation has terminated and produced a result, the `FlowResult` is notified of the value, which resumes any blocked and/or suspended readers. The rest of this section describes various schedulers that concurrently execute the submitted DataFlow computations.

Generally, computations are submitted to the scheduler as unevaluated by-name computations. Since any scheduling involves expensive resource and thread management, a scheduler needs to be properly shut down when the program terminates. The shutdown process of a scheduler can be a lengthy operation, thus its completion must be awaited. The interface is specified in 4.7.

```
trait Scheduler {
  def execute(f: => Unit): Unit
  def shutdown: BlockingAwait
}
```

Listing 4.7: Scheduler Interface

Based on the DataFlow model, computations are concurrently executed in nondeterministic order. Computations may have dependencies to other computations based on shared dataflow variables. Since these dependencies do not cause the computations to block until the data is available (they suspend instead), the computations can be considered as independent tasks and scheduled according to the *Parallel Task Pattern* [MSM09]. This allows each DataFlow computation to be submitted to a central scheduler, which executes the computations as tasks on a pool of threads.

The first scheduler developed uses the `ThreadPoolExecutor` (package `java.util.concurrent`), which provides methods to submit instances of `Runnable` and `Callable` for execution. `Callable`s can return a value which can be awaited by blocking the thread. Since this behaviour impedes suspending behaviour, as specified in `SuspendingAwait`, the tasks are submitted as `Runnable`s that return `Unit`. The `ThreadPoolExecutor` is configured to manage a pool of fixed size. Dynamic adaption of the pool size is not required, as tasks are expected to suspend instead of retaining the thread by blocking. The tasks are submitted to the `ThreadPoolExecutor` via an unbounded queue, which will not block attempts to schedule computations.

The second scheduler is a more comfortable variation of the first one: the thread pool is initialized with *Daemon Threads*, which are automatically disposed when the

main program terminates. This allows the programmer to omit an explicit scheduler shutdown.

Finally, the third scheduler uses the *Fork/Join Framework* [Lea00] which is targeted for Java 7. It is designed to support efficient implementation of concurrent divide-and-conquer algorithms, which fork sub-tasks (divide) and later join them (conquer) to calculate the overall result. As such the pool is optimized for tasks that spawn sub-tasks. Furthermore the `ForkJoinPool` is an optimized thread pool, which employs a work-stealing mechanism to efficiently distribute submitted tasks among its pool threads. The pool also provides an asynchronous mode for event-style tasks that are never joined. Since DataFlow computations are suspended and resumed by submitting new sub-tasks, and tasks never join forked sub-tasks, the `ForkJoinPool` in asynchronous mode is a perfect fit for ScalaFlow.

The scheduler is the central component in ScalaFlow which is accessed by all components that submit tasks. DataFlow variables and channels will resume computations by submitting their captured continuations to the scheduler. Instead of explicitly passing the scheduler to every component, the scheduler can be implicitly passed. Scala features *Implicit Parameters* [Ode10, p.103] which are automatically supplied. The compiler uses variables with a matching type which are in scope and explicitly marked as implicit e.g. `implicit val foo: String = "I'm implicitly supplied"`. This feature is used throughout the following components where access to the scheduler is required. Since it is still possible to explicitly supply values to implicit parameters, multiple schedulers can be used and explicitly passed to components. The final implementation of the `flow` method defines an implicit parameter to obtain a reference to the scheduler (see listing 4.8).

```
object Flow {
  // factory method to create a Flow and convert the by-name
     computation to a function
  def create[R](comp: => R @suspendable)(implicit scheduler:
     Scheduler): Flow[R] =
    new Flow[R](() => comp)

  // DSL method to create a flow, schedule it for execution and
      return the future result
  def flow[R](comp: => R @suspendable)(implicit scheduler:
     Scheduler): FlowResult[R] =
    create[R](comp).execute.result
}
```

```
final class Flow[A](val comp: () => A @suspendable)(implicit
    val scheduler: Scheduler) {
  val result = new FlowResult[A]

  def execute: Flow[A] = {
    scheduler execute { reset {
      result set comp.apply
    }}
    this
  }
}
```

Listing 4.8: Final Implementation of the Flow component

In conclusion: a `Flow` component has been implemented that captures computations as by-name parameters. The computations are applied to `reset` to enable delimited continuations and are then submitted as tasks to a central scheduler, which accessed via an implicit parameter. The `Flow` component returns a `FlowResult` for each DataFlow computation, which represents the flow's future result. The future can be awaited both by blocking and suspending operations. Respectively, obtaining the future's result is supported by a blocking as well as a suspending operation. Three different schedulers have been implemented that use different thread pools to implement the *Parallel Task Pattern*. The last scheduler utilizes the highly efficient `ForkJoinPool`.

4.3. Single-Assignment Variables

This section will incrementally describe the design of DataFlow variables. DataFlow variables are single-assignment: they can be bound to a value only once. Attempts to rebind the variable result in failure and undefined behaviour (practically, an exception is thrown). Readers attempting to access an unbound variable are suspended until the variable is bound. Variables may be concurrently accessed by multiple parallel computations. As a result monitor synchronization or atomic operations are required to provide thread-safe access. The following design uses a lock-free approach via atomic references and explicit data-race detection.

Variable state is expressed by an `Option[A]`, which is `None` when unbound and `Some(x)` when bound, where `x` denotes the bound value. To atomically change the variable's state without synchronization overhead, an `AtomicReference` from package `java.util.concurrent.atomic` is used. It provides the atomic `compareAndSet`

method, which compares the reference to a comparator and then sets the reference to a new value if the comparison succeeds. Thus binding the variable to a value is a matter of `AtomRef.compareAndSet(None, Some(value))`. If the variable is still unbound in the exact moment of binding it, the comparison will succeed and the `AtomicReference` is set to `Some(value)` to reflect the bound value. If the variable has already been bound, the comparison will fail and the implementation must deal with the failure by throwing an Exception.

As discussed in section 4.1, Scala's *Delimited Continuations* are used to suspend computations without blocking a thread. The reader of a `Variable[A]` is suspended by applying the CPS-primitive `shift`, which captures the reader's continuation. The suspended reader is expressed by the continuation `A => Unit`. To resume the reader, the continuation is applied to the variable's value of type `A`. The suspended readers, represented by their captured continuations, are stored in a FIFO-queue that allows concurrent access. This is an instance of `ConcurrentLinkedQueue` from package `java.util.concurrent`, which employs an efficient wait-free algorithm.

To resume the suspended readers, the readers are iteratively polled from the queue. Each reader is submitted to the scheduler as a task that applies the reader's continuation with the variable's new value.

The first implementation of a variable is demonstrated in listing 4.9. The `get` method suspends the reader by `shift`ing its continuation if the `Option[A]` in the `AtomicReference` is `None`. Method `set` tries to set the `AtomicReference` to `Some(value)`. If the atomic `compareAndSet` succeeds, all suspended readers are resumed. A failed `compareAndSet` indicates that the single-assignment variable has already been bound: an exception is thrown to signal failure.

```
class Variable[A](implicit val scheduler: Scheduler) {
  type Reader = (A => Unit)
  val suspendedReaders = new ConcurrentLinkedQueue[Reader]
  val v = new AtomicReference[Option[A]](None)
  def get: A = {
    if v.get.isDefined
      v.get.get
    else shift { k: Reader =>
      suspendedReaders offer k
    }
  }
  def set(v: A): Unit = {
    if v.compareAndSet(None, Some(v))
      resumeReaders()
```

```
    else
      throw new Exception("variable already bound")
  }
  def resumeReaders(v: A): Unit = {
    while (!suspendedReaders.isEmpty) {
      val k = suspendedReaders.poll
      if (k != null)
        scheduler execute { k(v) }
} } }
```

Listing 4.9: First implementation of a variable

The implementation presented in (4.9) has one major flaw: vulnerability to data-races. The interleaving of the concurrent execution of `get` and `set` is demonstrated in the following table; the suspended reader is enqueued after `resumeReaders` has completed. Therefore, the suspended reader is enqueued too late and will never be resumed.

concurrent `get`	concurrent `set`
`if defined == false`	
	`compareAndSet == true`
	`resumeReaders`
`shift`	
`suspendedReaders offer k`	

To solve this problem the implementation of `get` is extended to detect data-races (see listing 4.10). If the variable is bound after `shift`, `resumeReaders` might already have processed the queue. In this case the captured continuation is not enqueued. Instead it is immediately applied to resume the suspended reader, which is more efficient than submitting the continuation as a separate task to the scheduler. If no data-race is detected, the captured continuation is enqueued. After this, a second data-race check is done. If the variable is bound, `resumeReaders` is again potentially processing the queue. In this case `get` attempts to remove the continuation from the queue. If the remove succeeds `resumeReaders` can not resume the continuation, while if the operation fails `resumeReaders` has already resumed it. Thus if remove succeeds the continuation is immediately applied to resume the reader, ensuring that the reader is always resumed.

```
class Variable[A] {
  type Reader = (A => Unit)
  val suspendedReaders = new ConcurrentLinkedQueue[Reader]
  val v = new AtomicReference[Option[A]](None)
  def get: A @cps[Unit] = {
    if v.get.isDefined
      v.get.get
    else shift { k: Reader =>
       // handle DataRace #1
      if (v.get.isDefined)
        k(v.get)
      else {
        suspendedReaders offer k
        // handle DataRace #2
        if (v.get.isDefined && suspendedReaders remove k)
          k(v.get)
} } } }
```

Listing 4.10: Data-Race Safe Variable Implementation

Based on DataFlow variables it is possible to define signals, that suspend listeners until invoked. A signal does not transport any value, rather it represents a condition. Computations that await the signal are suspended. To transparently suspend and resume listeners, the signal uses a DataFlow variable and implements the `SuspendingAwait` trait to provide the suspending `sawait` method. Since `sawait` returns a `Boolean`, a `Variable[Boolean]` is used for its implementation. Because the signal is based on a single-assignment variable, the signal can only be invoked once. After invocation the variable is bound and will not suspend any further listeners. The implementation is listed in 4.11.

```
class Signal extends SuspendingAwait {
  val v = new Variable[Boolean]
  def invoke: Unit = v set true
  def sawait: Boolean @suspendable = v.get
}
```

Listing 4.11: A Signal based on a `Variable[Boolean]`

4.4. Channels

This section incrementally designs and implements the DataFlow channel, which is used for inter-flow communication. Starting with an *unbound channel*, the channel is

incrementally improved to support concurrent access of multiple consumers. Based on the unbound channel, a flow-control mechanism is introduced that suspends and resumes *producers*, resulting in the *bounded channel*. Methods to `terminate` a channel are also developed. Finally the channel is split into different traits for separation of concerns. The traits define DSL-aliases and the channel's interface is enriched with multiple higher-order functions to support Scala's for-comprehension.

4.4.1. Basic Unbounded Channel

A DataFlow channel has the semantics of a *queue*: elements are enqueued and dequeued in FIFO order. Multiple concurrent consumers compete for the queue's first element which is removed from the channel. Concurrent producers compete for the order in which elements are enqueued.

The elements are represented by a stream of DataFlow variables, implemented as a linked list. There are two references to the list: a reference to the first element allows efficient dequeueing, while a reference to the last element allows efficient enqueueing. The linked list is represented by the `Stream[A]` class, whose `value` field holds a DataFlow variable that suspends consumers if the channel is empty. The implementation is listed in 4.12.

```
class Channel[A] {
  class Stream[A] {
    val value = new Variable[A]
    var next: Stream[A] = null
  }
  var streamTake = new Stream[A]
  var streamPut  = streamTake
}
```

Listing 4.12: Linked list `Stream` to store the channel's elements

The following invariants must hold for the `Stream` of elements:

1. Reference `streamTake` points to the first element and must not be `null`
2. Reference `streamPut` points to the last element and must not be `null`
3. The last element's `value` is *unbound*, its `next`-reference is `null`
4. Any element with a bound `value`, has a non-null `next`-reference

The channel is concurrently accessed by consumers and producers. To synchronize access, the implementation adapts the "Two-Lock Concurrent Queue Algorithm" [MS96] which uses dedicated locks for enqueueing and dequeueing. Since the code is rewritten by the CPS-transformation, the intended scope of a `synchronized` block is lost. Instead, the `ReentrantLock` from package `java.util.concurrent.locks` is used, which provides the same semantics as monitor locking. Its extended capabilities are not used in this framework.

The following implementation of the basic unbounded channel is listed in 4.13. As with variables, this initial implementation has known shortcomings that will be discussed and resolved.

The channel provides two core operations: method `take` dequeues the first element from the `Stream` and returns it. Method `put` enqueues an element, appending it to the end of the `Stream`. When initialized, a new channel is empty; its `streamTake` and `streamPut` references point to the same `Stream[A]` instance, whose `value` contains one unbound variable and `next`-reference is null. To synchronize concurrent access the two locks `lockTake` and `lockPut` are used.

Method `take` first acquires `lockTake` to exclude other consumers. The first element is referenced by `streamTake`. Reading its DataFlow variable suspends the consumer if the variable is unbound. When the consumer is resumed, the element's `next` reference is not null ($Invariant_4$). The `streamTake` reference is advanced to the next element. `lockTake` is then released and `take` returns the value.

Method `put`, applied with value `v`, first aquires `lockPut` to exclude other producers. The last element is referenced by `streamPut`. By $Invariant_3$, its `value` is unbound and its `next` is `null`. The `next` reference is set to a newly created `Stream` instance, *then* the DataFlow variable `value` is bound to `v`. The order of setting the `next` reference and then binding `value` is crucial to satisfy $Invariant_4$ for suspended consumers. The method then advances the `streamPut` reference to the new element and finally releases `lockPut`.

```
class Channel(implicit val scheduler: Scheduler)[A] {
  class Stream[A] {
    val value = new Variable[A]
    var next: Stream[A] = null
  }

  var streamTake = new Stream[A]
  var streamPut  = streamTake
```

```
  val lockTake = new ReentrantLock
  val lockPut  = new ReentrantLock

  def put(v: A): Unit = {
    lockPut.lock
    val newLast = new Stream[A]
    streamPut.next = newLast
    streamPut.value set v
    streamPut = newLast
    lockPut.unlock
  }

  def take(): A @suspendable = {
    lockTake.lock
    val v = streamTake.value.get // suspends
          streamTake = streamTake.next
    lockTake.unlock
    v
  }
}
```

Listing 4.13: Unbounded Channel - Version #1

However as locks are bound to threads, releasing `lockTake` in method `take` will usually fail. If the channel is empty the consumer will be suspended by the unbound DataFlow variable. The consumer is resumed by the DataFlow variable as a scheduled task, which is executed on an arbitrary thread. Thus the resumed consumer can not release `lockTake` which may have been acquired from a different thread (prior to suspension). Instead of using `ReentrantLock`s, `Semaphore`s with a resource count of one can be used, since acquiring and releasing a semaphore is thread-independent.

Even with this improvement the method `take` is defective when multiple consumers `take` values from the channel, given the framework's aim to suspend consumers instead of blocking their threads. If the channel is empty, the first consumer that acquires lock `lockTake` is suspended by the unbound DataFlow variable. The lock is not released until the consumer resumes, thus other consumers are blocked until the lock is released.

The following implementation will resolve the problem of consumers being blocked instead of suspended (see listing 4.14). To suspend each consumer that is accessing `take` on an empty channel, the consumer must be suspended by its own DataFlow variable. Therefore `take` has to create an empty `Stream` element for next consumer if the channel is empty. To synchronize access to the `Stream`'s `next` element, `take` has

to acquire `lockPut`. After the lock is acquired, `take` appends a new `Stream` element that will suspend the next consumer.

The following adapted invariants must hold for the `Stream` of elements:

1. `streamTake` points to the next element to be taken and must not be `null`
2. `streamPut` points to the next element to be bound and must not be `null`
3. The element referenced by `streamPut` is *unbound*
4. Any element with a bound `value`, has a non-null `next`-reference

As a result the `put` method must check whether empty `Stream` elements have already been appended by `take`. If empty elements already exist `put` will bind the first existing element to a value, instead of creating a new element. Since each consumer is suspended by its own DataFlow variable, `lockTake` can be released before reading the variable and potentially getting suspended. In this manner the framework avoids attempting to acquire and release `ReentrantLock` from different threads.

To conclude the discussion of locking: the `take` method acquires both locks if the channel is empty to exclude producers from concurrently appending. This could impair channel performance when producers compete with consumers for the `lockPut` on an empty channel.

```
class Channel[A](implicit val scheduler: Scheduler) {
  class Stream[A] {
    val value = new Variable[A]
    var next: Stream[A] = null
  }

  var streamTake = new Stream[A]
  var streamPut  = streamTake

  val lockTake = new ReentrantLock
  val lockPut  = new ReentrantLock

  def put(v: A): Unit = {
    lockPut.lock
    val dfvar = streamPut.value

    if (streamPut.next == null)
      streamPut.next = new Stream[A]

    streamPut = streamPut.next
    lockPut.unlock
```

```
    dfvar set v
  }

  def take(): A @suspendable = {
    lockTake.lock

    // double-check whether there is no next element
    if (streamTake.next == null) {
      lockPut.lock
      if (streamTake.next == null)
        streamTake.next = new Stream[A]
      lockPut.unlock
    }

    val dfvar = streamTake.value
          streamTake = streamTake.next
    lockTake.unlock
    dfvar.get //suspends
  }
}
```

Listing 4.14: Unbounded Channel - Version #2

4.4.2. Limiting Producers and Stopping Consumers

This section extends the unbounded channel to a *bounded channel* with a specific capacity. Producers are allowed to append values to the channel until the channel's capacity is reached. Producers that call `put` on a full channel are suspended. They are resumed in FIFO order when the channel is drained by consumers. The unbounded channel described in the previous section implicitly operates in `eager` mode, meaning that its producers are executing unconstrained. The channel is also extended with a termination mechanism, which is used to prevent consumers from waiting indefinitely for new elements. The bounded channel supports multiple types of flow-control which are determined by the channel's capacity:

Capacity	Flow-Control Type	Semantics
`0`	Lazy	Producers get suspended by default and are resumed on consumer's demand.
Infinite (`Int.MaxValue`)	Eager	No restriction/suspension of producers, producers are able to consume all available resources.
`Int`	Bounded Buffer	Producers are allowed to fill the channel up to its capacity. If full, producers are suspended.

To suspend producers the CPS primitive `shift` is used. The operator captures the producer's continuation as a function of type `Unit => Unit`. In contrast to consumers, the producer does not need any input to resume producing, thus the function's parameter is `Unit`. The captured continuation is enqueued to a concurrent queue, since consumers concurrently access it to resume producers. To keep track of the channel's queue length, an `AtomicInteger` is used, which allows atomic increment and decrement operations. The length is incrementing when the `put` method enqueues an element and decrementing when the `take` method dequeues an element.

So far, channels are not able to signal termination to consumers if the producers finished producing. Consumers are suspended by the empty channel and will not be resumed until another element is put into the channel. To resume consumers on termination, the unbound DataFlow variables in `Stream` that suspend the consumers are bound to the bottom value `null`, which is expressed by `val T = null.asInstanceOf[A]`. This requires the `take` method to check whether the value is `T` (following resumption) and react accordingly.

Signalling termination to consumers could be specified by an explicit type for method `take`'s return value: `def take: Option[A] @suspendable`, where `Option[A]` indicates that the channel returns a `None` if the channel is terminated and empty. However this would force consumers to pattern-match on the result of `take` even if the channel is never terminated. This implementation is rejected since signalling the termination is exceptional and only occurs once, if the channel is explicitly terminated and if the channel is empty. Instead the implementation will throw a special `TerminatedChannel` exception. This allows concise consumer code that handles channel termination in a `catch` block. Furthermore, the exception can be completely hidden

from consumers by providing higher-order functions such as `foreach` and `map`, which are developed in the next section. To mitigate the performance impact of throwing exceptions, `TerminatedChannel` mixes-in the trait `ControlThrowable`. This avoids costly stack trace generation by using a singleton instance of `TerminatedChannel`, rather than constructing one for every throw.

The channel's termination state is implemented by `@volatile var terminatedFlag: Boolean`. The `@volatile` modifier forces the JVM to access main memory on field read/write, ignoring thread-local caches, thus its value is immediately visible to all threads. This is used in both the `put` and `take` methods: `take` immediately throws an exception if the channel is already terminated which short-circuits the code that would otherwise acquire `lockTake`. Similarly, `put` checks the flag and throws the `TerminatedChannel` exception if the channel is terminated and empty, again short-circuiting the rest of the operation.

The implementation of the `put` method and all relevant fields of the channel class are listed in 4.16. This is only a slight extension to the previous implementation (see listing 4.14). After `lockTake` is acquired, new `Stream` elements are only appended if the channel has not yet been terminated. After `streamNext` is advanced to the next element, the counter is decremented to reflect the channel's new size. If the channel is empty, the counter will be decremented to a negative counter, which represents waiting consumers. Finally the first suspended producer is resumed (in FIFO order) by invoking the `resumeReaders` method, after which the value returned from the DataFlow variable is checked for `T`, which signals a resumption due to termination of the channel.

```
class Channel[A](val capacity: Int)(implicit val scheduler:
   Scheduler) {
  class Stream[A] {
    val value = new Variable[A]
    var next: Stream[A] = null
  }

  type Producer = (Unit => Unit)

  var streamTake = new Stream[A]
  var streamPut  = streamTake

  val lockTake = new ReentrantLock
  val lockPut  = new ReentrantLock

  val suspendedProducers = new ConcurrentLinkedQueue[Producer]
```

```
  val counter = new AtomicInteger(0)

  @volatile var terminatedFlag = false

  val T = null.asInstanceOf[A]

  def take(): A @suspendable = {
    if (terminatedFlag && streamTake.next == null)
      throw TerminatedChannel

    lockTake.lock
    if (!terminatedFlag && streamTake.next == null) {
      lockPut.lock
      if (streamTake.next == null)
        streamTake.next = new Stream[A]
      lockPut.unlock
    }

    val dfvar = streamTake.value

    if (streamTake.next != null) {
      streamTake = streamTake.next
      counter.decrementAndGet
    }
    lockTake.unlock

    resumeProducer()

    val v = dfvar.get // suspend
    if (v == T)
      throw TerminatedChannel
    v
  }

  def resumeProducer(): Unit = {
    val k = suspendedProducers.poll
    if (k != null)
      scheduler execute { k() }
  }
}
```

Listing 4.15: Bounded Channel with Termination - `take`

The implementation of the `put` method is listed in 4.16. The channel's fields are omitted since they are already listed in 4.15. The method begins by checking for termination after which the counter is incremented. The producer is suspended if the counter is over capacity. The producer must then acquire the first `lockPut` lock.

If a consumer `take`s an element from the channel and thus resumes producer P_a, another producer P_b in method `put` might detect a data race and continue execution. Both producers will have to acquire the first `lockPut` lock which synchronizes their execution. The producer that is able to acquire the lock rechecks the counter and suspends if the channel is full. When suspending succeeds, the first lock is also released. The second lock protects the code that binds the `streamPut` reference and advances it. Producers that have acquired the first lock, already own the lock and increment the lock's hold count instead of acquiring it a second time. Therefore the method `put` ends by releasing the `lockPut` lock until the hold count is zero.

```
class Channel[A](val capacity: Int)(implicit val scheduler:
   Scheduler) {
  def isFull = (counter.get >= capacity)

  def put(v: A): Unit @suspendable = {
    if (terminatedFlag)
      throw new Exception("Channel is terminated")

    if (counter.incrementAndGet >= capacity)
      suspendProducer(None)
    else
      cpsunit

    // producer #1 detects data-race while suspending,
       continues.
    // consumer resumes producer #2, both will stop at the lock
       .
    // first wins

    lockPut.lock
    if (isFull)
      suspendProducer(Some(lockPut))
    else
       cpsunit

    lockPut.lock
    val dfvar = streamPut.value

    if (streamPut.next == null)
      streamPut.next = new Stream[A]
    streamPut = streamPut.next

    // release all locks
    while (lockPut.isHeldByCurrentThread)
      lockPut.unlock

    dfvar set v
```

```
  }

  def suspendProducer(lock: ReentrantLock): Unit @suspendable =
    shift { k: Producer =>
      if (!isFull) // handle Data-Race #1
        k()
      else {
        suspendedProducers offer k
        if (!isFull && suspendedProducers.remove(k)) // handle
            Data-Race #2
          k()
        else
          lock.unlock
      }
    }
```

Listing 4.16: Bounded Channel with Termination - `put`

To signal termination of the channel and resume suspended consumers, the method `terminate` is defined (see listing 4.17). The channel stores its termination state in the `@volatile` field `terminatedFlag`. Additionally a `terminatedSignal` signals termination. The method `terminate` obtains exlusive access to the `Stream` elements by acquiring both locks. The next element that would be bound by `put` is referenced by `streamPut`. It is bound to the bottom termination value `T` to resume its suspended consumer. Since consumers may have created additional empty DataFlow variables in method `take`, all elements reachable from `streamPut` are also bound to `T` to resume their consumers. Finally the locks are released and `terminatedSignal` is invoked to signal termination of the channel.

```
class Channel[A](val capacity: Int)(implicit val scheduler:
   Scheduler) {
  val lockTake = new ReentrantLock
  val lockPut  = new ReentrantLock
  val suspendedProducers = new ConcurrentLinkedQueue[Producer]

  @volatile var terminatedFlag = false
  val terminatedSignal = new Signal

  def isTerminated: Boolean = terminatedFlag
        def terminated: SuspendingAwait = terminatedSignal

  def terminate: Unit = {
    if (terminatedFlag)
      throw new DataFlowException("Channel already terminated")

    terminatedFlag = true
```

```
    lockTake.lock
    lockPut.lock

    streamPut.value := T

    while (streamPut.next != null) {
      streamPut = streamPut.next
      streamPut.value := T
    }

    lockPut.unlock
    lockTake.unlock

    terminatedSignal.invoke
  }
```

Listing 4.17: Bounded Channel with Termination - `terminate`

In conclusion, the bounded channel is able to operate in three flow-control modes: eager, lazy and as a bounded buffer. The channel supports termination, which resumes consumers and then stops them by throwing an exception. This completes implementation of the basic channel functionality.

4.4.3. Enriching the Interface

This section splits the bounded channel into different traits for separation of concerns. The channel's interface will be enriched with DSL aliases and multiple higher-order functions to support Scala's for-comprehension.

The channel provides three basic operations: putting a value into a channel, taking a value from a channel and terminating a channel. Therefore, the interface is split into the traits `ChannelPut` (4.18), `ChannelTake` (4.19) and `ChannelTerminate` (4.20). Each trait defines the basic abstract methods to deal with its concern and additional DSL aliases.

```
trait ChannelPut[A] { self: Channel[A] =>
  def put(value: A): Unit @suspendable
  def <<(v: A) : Unit @suspendable = put(v)

  def putAll(values: Iterable[A]): Unit @suspendable
  def <<<(vs: Iterable[A]) : Unit @suspendable = putAll(vs)
}
```

Listing 4.18: Trait `ChannelPut`

```
trait ChannelTake[A] { self: Channel[A] =>
  def take: A @suspendable
  def apply(): A @suspendable = take
}
```

Listing 4.19: Trait `ChannelTake`

```
trait ChannelTerminate[A] { self: Channel[A] =>
  def isTerminated: Boolean
  def terminated: SuspendingAwait

  def terminate: Unit
  def <<# : Unit = terminate
}
```

Listing 4.20: Trait `ChannelTerminate`

The for-comprehension, as described in section 3.3, expands `for` into applications of the higher-order functions `foreach`, `map`, `flatMap` and `filter`. Thus, these functions are implemented first (see listing 4.21).

```
trait ChannelTake[A] { self: Channel[A] =>
  def take: A @suspendable
  def apply(): A @suspendable = take

  def foreach[U](f: A => U @suspendable): Unit @suspendable
  def map[M](f: A => M @suspendable): Channel[M]
  def flatMap[M](f: A => Channel[M] @suspendable): Channel[M]
  def flatMapIter[M](f: A => Iterable[M]): Channel[M]
  def filter(p: A => Boolean @suspendable): Channel[A]
}
```

Listing 4.21: Trait `ChannelTake` with higher-order functions

The basic pattern for looping over a channel's elements is to use an unrestricted `while (true)` loop inside a `try-catch` block to intercept the `TerminatedChannel` exception. The loop successively takes values from the channel by calling the `take` method. When the channel is terminated and empty, the `put` method will throw the `TerminatedChannel` exception which breaks the `while` loop. The exception is intercepted and silently discarded, after which the looping method terminates and returns control to the caller. Further applications of the method on the terminated channel instantly return because the channel will throw the `TerminatedChannel` exception on the first call to `take`.

The `foreach` method takes a function that is successively applied to each element in the channel (see 4.22). The passed function's return type is in general `Unit`, since the function is meant to perform computations with side effects rather than producing a result. Generalizing from this behaviour, `foreach` accepts any function of type `A => U @suspendable`, where `U` implies the `Unit` type but does not enforce it. Nevertheless, `foreach` returns `Unit` after completion. The method does not run concurrently: control is only returned to the caller when `foreach` has completed.

```
class Channel[A] extends ChannelTake[A] {
  def foreach[U](f: A => U @suspendable): Unit @suspendable = {
    val ch = this
    try while(true)
      f( ch.take )
    catch { case TerminatedChannel => }
} }
```

Listing 4.22: Channel implementing the higher-order-function `foreach`

Since the CPS-transformation transforms the `while` loop into recursive function calls, code using `while(true)` such as the above will eventually produce a `StackOverflowError`. This can be avoided by using channels with a restricted capacity to increase the chance of consumers getting suspended. When resumed the consumers are submitted as a new task with clean stack, minimizing the chance of producing a `StackOverflowError`.

However a cleaner solution is to use *Trampolining* [Dou09, Ode08]: a recursive function does not call itself but returns a function (thunk) which is to be called next. The implementation is listed in 4.23. A function either returns a `Call` containing the `thunk` function that is to be executed next or it returns a `Return` which stops the recursion and returns the embodied value (see listing 4.23). Therefore the function always returns and thus unwinds the stack.

```
sealed trait TailRec[A]
case class Return[A](result: A) extends TailRec[A]
case class Call[A](thunk: () => TailRec[A] @suspendable)
   extends TailRec[A]

def tailrec[A](comp: TailRec[A]): A @suspendable = comp match {
  case Call(thunk) => tailrec( thunk() )
  case Return(x)   => x
}
```

Listing 4.23: Trampolining

Based on `tailrec` a `loop` operator can be implemented that executes passed expressions indefinitely. The operator `loop` takes a by-name computation `comp` which returns type `U`, indicating but not enforcing the `Unit` type. The computation represents the loop's body. Since the computation does not return `TailRec`, the inner function `f` wraps `comp` and returns a recursive `Call` thunk after the computation has been executed (see listing 4.24).

```
def loop[U](comp: => U @suspendable): Unit @suspendable = {
  def f(): TailRec[Unit] @suspendable = {
    comp
    Call( () => f() )
  }
  tailrec( f() )
}
```

Listing 4.24: Implementation of an infinite loop using Trampolining

A constrained loop is implemented by the `loopWhile` operator which is listed in 4.25. The operator is written in curried syntax to provide natural application: `loopWhile(x != y) {...}`. The constraint `c` is passed as a by-name parameter resulting in `Boolean`. The constraint is checked before executing the computation. If the constraint fails, `Return` with the `Unit`-value `()` is returned to stop `tailrec`.

```
def loopWhile[U](c: => Boolean)(comp: => U @suspendable): Unit
   @suspendable = {
  def f(): TailRec[Unit] @suspendable = {
    if (c) {
      comp
      Call( () => f() )
    } else Return( () )
  }
  tailrec( f() )
}
```

Listing 4.25: Implementation of a constrained loop using Trampolining

Using `for` in conjunction with the `yield` keyword expands the loop to applications of `map` and `flatMap`. The implementation of `map` and `flatMap` take a function `f` that represents the `yield` expression. Their implementation is listed in 4.26 and 4.27, respectively.

The `map` method, applied on channel C_a, creates a new channel C_b which is immediately returned to the caller. Concurrently, `map` transforms elements from C_a by

applying the passed function `f` to each element. The transformed elements are successively put into the new channel C_b. `map` does not have a `@suspendable` annotation on its return-type because it instantly returns the new channel without suspending.

```
class Channel[A] extends ChannelTake[A] {
  def map[M](f: A => M @suspendable): Channel[M] = {
        val inCh  = this
        val outCh = Channel.createLike[M](inCh)

        scheduler execute { reset {
          try loop { outCh put f( inCh.take )   }
          catch { case TerminatedChannel => outCh.terminate }
        }}
        outCh
} }
```

Listing 4.26: Channel with higher-order function `map`

The `flatMap` method is similar, in that it concurrently executes the transformation and instantly returns the new channel. However its passed function `A => Channel[M] @suspendable` returns a `Channel` instead of a single element. Therefore the resulting channels are concatenated: each element of the returned `Channel[M]` is separately put into the new channel (flattening).

```
class Channel[A] extends ChannelTake[A] {
  def flatMap[M](f: A => Channel[M] @suspendable): Channel[M] =
      {
    val inCh  = this
    val outCh = Channel.createLike[M](inCh)

    scheduler execute reset {
      def transfer(from: Channel[M], to: Channel[M]): Unit @
          suspendable = {
        try loop { to put from.take }
        catch { case TerminatedChannel => }
      }
      try while(true)
        transfer(f( inCh.take ), outCh)
      catch { case TerminatedChannel => outCh.terminate }
    }
    outCh
} }
```

Listing 4.27: Channel with higher-order function `flatMap`

For-comprehensions of type `for (x <- xs; if x > y)` are also expanded into applications of `filter` or `withFilter`. The constraint is passed as a predicate of

type `A => Boolean @suspendable`. While each application of `filter` returns a new `Channel` containing only elements satisfying the predicate, `withFilter` is intended to return a new object that accumulates the predicates and transparently applies them on successive calls of `foreach`, `map` and `flatMap`. This avoids the need to create a separate channel for each filter constraint.

The `filter` method is implemented in a similar manner to `map` (see listing 4.28). When applied to channel C_a, it creates a new channel C_b which is immediately returned. Concurrently, values taken from C_a and satisfying the predicate are put into C_b.

```
class Channel[A] extends ChannelTake[A] {
  def filter(p: A => Boolean @suspendable): Channel[A] = {
    val inCh  = this
    val outCh = Channel.createLike[A](inCh)

    scheduler execute { reset {
      try loop {
        val in = inCh.take
        if (p(in)) outCh put in
        else       cpsunit
      } catch { case TerminatedChannel => outCh.terminate }
    }}
    outCh
} }
```

Listing 4.28: Channel with higher-order function `filter`

By contrast the `withFilter` method creates a new instance of `ChannelFilter` that stores the filter predicate (see listing 4.29). `ChannelFilter` implements variations of `foreach`, `map` and `flatMap` that additionally check the predicate. To accumulate predicates, `ChannelFilter` also implements `withFilter` which creates a new instance of `ChannelFilter` that combines the existing predicate with the new predicate by logical conjunction.

```
class Channel[A] extends ChannelTake[A] {
  def withFilter(p: A => Boolean @suspendable): ChannelFilter =
    new ChannelFilter(p)

  class ChannelFilter(p: A => Boolean @suspendable) {
    def foreach[U](f: A => U @suspendable): Unit @suspendable =
        {
      val ch = self
      try loop {
        val v = ch.take
```

```
        if (p(v)) f(v)
        else      cpsunit
      } catch { case TerminatedChannel => }
    }
    def map[M](f: A => M @suspendable)(implicit scheduler:
       Scheduler): Channel[M] = { ... }
    def flatMap[M](f: A => Channel[M] @suspendable): Channel[M]
        = { ... }
    def withFilter(q: A => Boolean): ChannelFilter =
      new ChannelFilter(x => p(x) && q(x))
} }
```

Listing 4.29: Channel with higher-order function `withFilter`

Additional higher-order functions are implemented, with the following semantics:

- `def fold[B](z: B)(f: (A,B) => B @suspendable): B @suspendable`
  Applies the binary operator `f` to the start value `z` and all elements of this channel, from first element to the last, suspends the caller if channel is empty.
- `def reduce[B >: A](f: (A,B) => B @suspendable): B @suspendable`
  Applies the binary operator `f` to all elements of this channel, from first element to the last, suspends the caller if channel is empty.
- `def duplicate(): (Channel[A], Channel[A])`
  Concurrently puts all elements of this channel into two new channels.
- `def split(p: A => Boolean @suspendable): (Channel[A], Channel[A])`
  Concurrently splits this channel into two new channels, elements satisfying the predicate `p` are put into the first channel, all other elements are put into the second channel.
- `def combine[B <: A](ch2: ChannelTake[B]): Channel[A]`
  Concurrently puts all elements of this channel and of `ch2` into a new channel: the order of elements is not guaranteed. The new channel will not be terminated until both source channels are terminated and fully processed
- `def concat[B <: A](ch2: ChannelTake[B]): Channel[A]`
  Concurrently puts all elements of this channel and *then* all elements of `ch2` into a new channel and *then* terminates the new channel.
- `def transfer[B](to: ChannelPut[B], f: A => B): Unit`
  Concurrently transfers all elements of this channel into the given channel `to`, does not terminate.

4.5. Pipeline

A sequence of concurrent computations, where each computation feeds its output into the next computation is called a *Pipeline*. The individual computations represent the pipeline's *Stages*. This section demonstrates the construction of a pipeline by concatenating channels using the higher-order function `map`. Finally a fluent interface is defined, to simplify pipeline construction.

A stage is a concurrent computation that connects two channels. Using `map`, the passed transformation function is scheduled for concurrent execution and a new channel is immediately returned. Thus stages can be defined by applications of `map` on the input channel. The returned channel is the stage's output and the input to the next stage. A two-staged pipeline can be created by two applications of `map`, as demonstrated in listing 4.30.

```
val ch1 = Channel.create[Int]
def stage1(x: Int): Int @suspendable = 2*x
val ch2 = ch1.map( stage1 )
def stage2(x: Int): Double @suspendable = x + 0.1
val ch3 = ch2.map( stage2 )
flow { ch1.put(1); ch1.put(2); }
flow { ch3.foreach(d => println(d)) }
```

Listing 4.30: Creating a pipeline just by using `map`

To create pipelines with a fluent interface [FE], ScalaFlow provides the `Pipeline` class, which represents the pipeline's source. Its implementation is listed in 4.31. This class contains an internal channel which is the pipeline's source, with associated methods to `put` values into the channel and to `terminate` it. Thus the internal channel contains the input values for the first stage. A stage is created by applying the method `stage` with a transformation function `f: A => B @suspendable`. It returns a `PipelineStage[A,B]` that uses the `Pipeline`'s internal channel as its input channel. The stage applies `map` on the input channel which creates the stage's output channel. The `PipelineStage` also implements the `stage` method which creates the next stage. The next stage is in turn supplied by the previous stage's output channel. In contrast to the `Pipeline` class, `PipelineStage` does not provide methods to `put` values or `terminate`. Putting values into the stage and propagating termination is handled by `map`. The last call to `stage` creates a stage that represents the pipeline's sink. Thus every `PipelineStage` provides a `foreach` method to process its output channel.

```
class Pipeline[A](implicit val scheduler: Scheduler) {

  val inCh = Channel.create[A]

  def put(v: A): Unit @suspendable =
    inCh put v

  def terminate: Unit @suspendable =
    inCh.terminate

  def stage[B](f: A => B @suspendable): PipelineStage[A,B] =
    new PipelineStage[A,B](inCh, f)
}


class PipelineStage[A,B](val inCh: Channel[A], f: A => B @
   suspendable)(implicit val scheduler: Scheduler) {

  val outCh: Channel[B] = inCh.map(f)

  def stage[C](f: B => C @suspendable) =
    new PipelineStage[B,C](outCh, f)

  def foreach(f: B => Unit @suspendable): Unit @suspendable =
    outCh.foreach(f)
}
```

Listing 4.31: Pipeline with stages implemented by `map`

4.6. Conclusion

To summarise the design of ScalaFlow, a class diagram is presented in 4.1. The `Scheduler` is the central component of the framework, with three different implementations. The `Channel` is a realisation of three different traits, which define the DSL aliases and the higher-order functions. The `Signal` class is based on `Variable` and the `Pipeline` class is implemented by using `Channel`'s higher-order function `map` to represent the concurrent `PipelineStage`s.

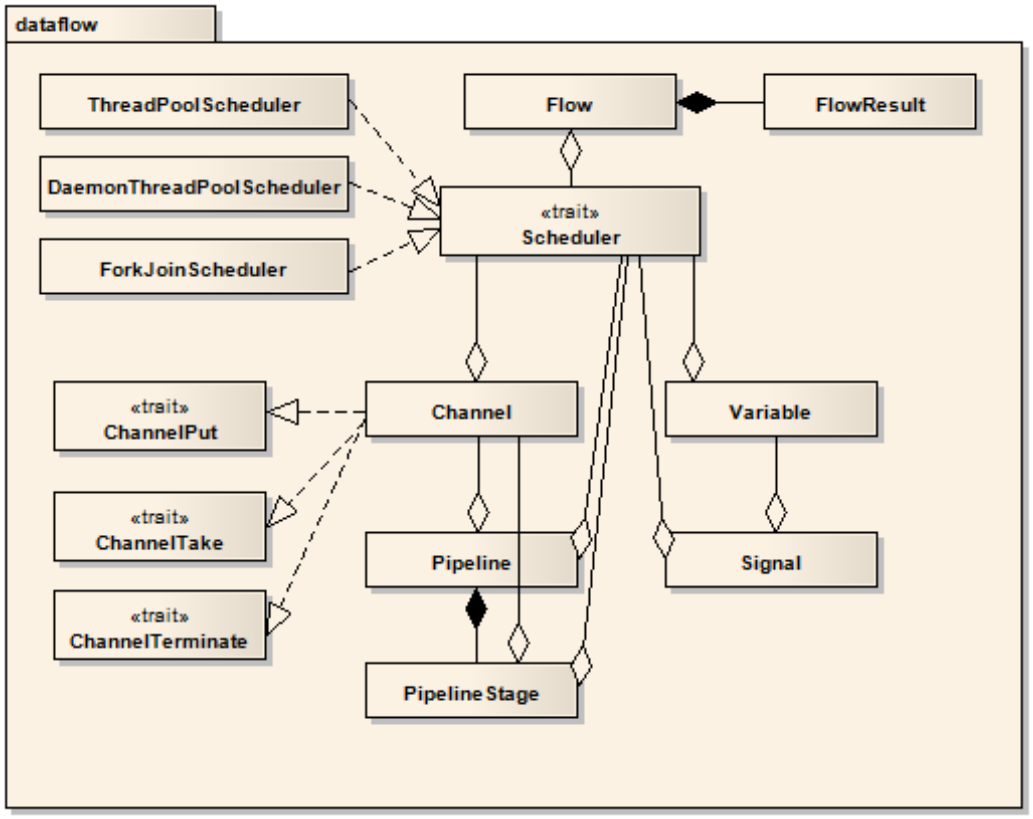

Figure 4.1.: Class diagram of ScalaFlow

5

Networking Capabilities

This chapter extends ScalaFlow with networking capabilities to communicate with remote nodes and to create distributed systems. As discussed earlier, a major design goal of the framework is the avoidance of thread blocking operations. This rules out use of the standard Input/Output (I/O) operations in the `java.io` package, which are realized by thread-blocking `Stream` abstractions. Since Java 1.4.2, the `java.nio` package supports a new non-blocking I/O approach (NIO) based on multiplexing selectors, which is used in the following design for ScalaFlow's I/O subsystem.

NIO features `Channels` that represent connections to entities and are capable of performing I/O operations on files and sockets. Channels implementing the interface `SelectableChannel` can be registered with a `Selector` that monitors the readiness of any number of I/O operations, on every channel registered with the selector (multiplexing). The selector is polled to retrieve a set of `SelectionKeys`, where each key is bound to a specific channel and represents the readiness of its operations. Each key also holds an arbitrary 'attachment object' to allow for maintaining state or providing context to handlers. Key processing follows the *Reactor Pattern* [Sch94, Rot07] and imposes inversion of control on the code: a dispatcher polls the selector and processes the selected keys. Each `SelectionKey`'s ready operations are dispatched

to handlers which react to these extrinisic events. By contrast, programs using thread-blocking I/O operations are in control: their control flow advances after each blocking operation has finished processing the stream.

Section 5.1 will describe a lightweight event dispatcher based on Java's NIO library. Based on this dispatcher, section 5.2 will describe ScalaFlow network sockets. Their interface is based on DataFlow channels to read from and write to the socket. Accepting connections from client is implemented by the acceptor in section 5.3. Finally section 5.4 describes a HTTP server capable of processing HTTP requests and generating HTTP responses, based on socket read and write channels.

5.1. NIO Dispatcher

Listing 5.1 sketches the basic `dispatch`-loop: the method polls the singleton `Selector` instance for `SelectionKeys` using the blocking `select` method until the dispatcher is shutdown, which is marked by the volatile `_shutdown`-flag. Each `SelectionKey` is passed to `dispatchKey` which pre-processes the key's ready operations and dispatches to registered event handlers. To store distinct handlers for each operation, the handler functions are aggregated in the `NIO_Handler` class (see listing 5.2) which is stored as an attachment to every `SelectionKey`.

```
class Dispatcher {
  val selector = Selector.open
  @volatile var _shutdown = false
  def dispatch(): Unit = {
    while (!_shutdown) {
      if (selector.select > 0) {
        val it = selector.selectedKeys.iterator
        while (it.hasNext) {
          val key = it.next
          it.remove
          dispatchKey(key)
  } } } }
  def dispatchKey(key: SelectionKey): Unit = {
    val handler = key.attachment.asInstanceOf[NIO_Handler]
    if (key.isValid && key.isReadable)
      ...
} }
```

Listing 5.1: Basic Dispatching

```
class NIO_Handler {
  type OpHandler      = SelectableChannel => Unit
  type FailureHandler = Throwable         => Unit

  var onReadable: Option[OpHandler]  = None
  var onWritable: Option[OpHandler]  = None
  var onAccepted: Option[OpHandler]  = None
  var onConnected: Option[OpHandler] = None
  var onClosed: Option[OpHandler]    = None
  var onFailure: Option[FailureHandler] = None
}
```

Listing 5.2: Handler aggregating specific handler-functions

NIO provides four distinct operation flags to indicate readiness of a channel operation:

- `OP_ACCEPT` - the channel is ready to accept a new connection
- `OP_CONNECT` - the channel can complete the connection process
- `OP_READ` - the channel can be read without blocking
- `OP_WRITE` - the channel can be written to without blocking

Each operation flag is wrapped in the sealed class `NIO_OP` to maintain type safety. This class enriches the operation with methods to add interest and remove interest from a specified bitmask (see listing 5.3), which is used to register multiple interests at the channel's `SelectionKey`.

```
sealed class NIO_OP(op: Int) {
  def addInterestTo(ops: Int): Int       = ops | op
  def removeInterestFrom(ops: Int): Int  = ops & ~op
}
object Dispatcher {
  case object Accept  extends NIO_OP(SelectionKey.OP_ACCEPT)
  case object Connect extends NIO_OP(SelectionKey.OP_CONNECT)
  case object Read    extends NIO_OP(SelectionKey.OP_READ)
  case object Write   extends NIO_OP(SelectionKey.OP_WRITE)
}
```

Listing 5.3: Wrapping the raw operation-flags

A NIO channel must first be registered at the dispatcher using the `register` method, which optionally takes functions to handle closed connections and failures (see listing 5.4). Upon initial registration, an instance of `NIO_Handler` is created and attached

to the channel's key. Each operation of interest has to be specifically registered at the dispatcher using the `registerOpInterest` method, which takes a handler function that will be invoked to handle the operation. The `NIO_Handler`, which is attached to the key, is updated with the registered handler function and subsequently used by `dispatchKey` during dispatching.

```
class Dispatcher {
  def register(ch: SelectableChannel,
      onClosed: Option[NIO_Handler#OpHandler] = None,
      onFailure: Option[NIO_Handler#FailureHandler] = None):
         Unit = {

    val handler = new NIO_Handler
    handler.onClosed  = onClosed
    handler.onFailure = onFailure

    selectorGate.synchronized {
      selector.wakeup()
      ch.register(selector, 0, handler)
    }
  }

  def registerOpInterest(ch: SelectableChannel,
      op: NIO_OP,
      f: NIO_Handler#OpHandler): Unit = {

    selectorGate.synchronized {
      selector.wakeup()

      val ops    = ch.keyFor(selector).interestOps
      val newOps = op.addInterestTo(ops)

      val handler    = ch.keyFor(selector).attachment.
         asInstanceOf[NIO_Handler]
      val newHandler = updateHandler(handler, op, f)

      ch.register(selector, newOps, newHandler)
    }
}
```

Listing 5.4: Dispatcher operations to register interest in channel operations

This basic design directly dispatches readiness events to registered handlers and imposes *Inversion-of-Control* on the program. However, it is possible to use continuations to avoid this. The method `waitFor` takes a channel and the operation to wait for in curried syntax (see listing 5.5). The CPS primitive `shift` is used to capture the current continuation as a function of type `SelectableChannel => Unit`,

which corresponds to the type definition `OpHandler` from `NIO_Handler`. This type allows the continuation to be captured and directly registrated as a handler. When the handler is invoked by the dispatcher, the continuation is resumed and `waitFor` returns the channel for which an operation became ready (see 5.6).

```
class Dispatcher {
  import scala.util.continuations._

  ...

  def waitFor(ch: SelectableChannel)(op: NIO_OP):
     SelectableChannel @suspendable =
    shift { k: NIO_Handler#OpHandler =>
      registerOpInterest(ch, op, k)
    }
}
```

Listing 5.5: `waitFor` registering continuation as event-handler

```
flow {
  val ch: SelectableChannel
  val d: Dispatcher
  d.register(ch, None,None)

  val wait = d.waitFor(ch) _

  wait(Dispatcher.Connect)
  // continuation #1
  println("connectable")

  wait(Dispatcher.Read)
  // continuation #2
  println("readable")

  wait(Dispatcher.Write)
  // continuation #3
  println("writeable")

  // end of continuations
}
```

Listing 5.6: Using `waitFor`

5.2. Network Sockets

A *Socket* is a bidirectional endpoint for network communication. The NIO package provides a `SocketChannel` that supports non-blocking operations in conjunction with a `Selector`. This section will create a `Socket` that uses DataFlow `Channel`s for reading and writing to the socket. The socket provides a `connect` method that takes an `InetSocketAddress` to address the endpoint (see listing 5.7). The socket channel (`socketCh`) is connected to the address and registered at the dispatcher. Interest in the `Connect` operation is then registered using the `waitFor` method. As explained in section 5.1, `waitFor` captures the caller's continuation and registers the continuation as an event handler, which is invoked when the socket is connected. Upon successful connection, the socket invokes the dataflow signal `connectedSignal` to resume any computations waiting for the established connection.

```
class Socket(implicit val dispatcher: Dispatcher, implicit val
    scheduler: Scheduler) {

  val socketCh: SocketChannel // NIO
  val connectedSignal = new Signal

  def connected: SuspendingAwait = connectedSignal

  def connect(address: InetSocketAddress): Unit @suspendable =
  {
    socketCh.connect(address)
    dispatcher.register(socketCh, Some(onClosed), Some(
        onFailure))
    dispatcher.waitFor(socketCh)(Connect)
    connectedSignal.invoke
  }

  def onClosed(ch: SelectableChannel): Unit
  def onFailure(failure: Throwable): Unit
}
```

Listing 5.7: Socket - Connecting to an address

To read from or write to a socket, client code must call the socket's `process` method, to initiate processing of its read and write DataFlow channels (`readCh` and `writeCh`, see listing 5.8). This method returns a suspending future (`SuspendingAwait`) which is invoked when both the read channel has been terminated due to an end-of-stream on the network socket, and the write channel has been terminated by client code (signalling the end of data to be sent).

```
class Socket(implicit val dispatcher: Dispatcher, implicit val
    scheduler: Scheduler) {
  val socketCh: SocketChannel // NIO

  val readCh = Channel.create[Array[Byte]]
  val writeCh = Channel.create[Array[Byte]]

  // public interface
  def read: ChannelTake[Array[Byte]] = readCh
  def write: ChannelPut[Array[Byte]] = writeCh

  def process(): SuspendingAwait = {
    val readSignal  = processRead()
    val writeSignal = processWrite()
    new SuspendingAwait {
      def sawait: Boolean @suspendable = (readSignal.sawait &&
          writeSignal.sawait)
    }
  }

  def processRead(): SuspendingAwait
  def processWrite(): SuspendingAwait
}
```

Listing 5.8: Socket - Processing read and write channels

The method `processRead` returns a `SuspendingAwait` which is invoked when reading the socket results in an end-of-stream (see listing 5.9). Filling the read channel `readCh` with data read from the socket channel is scheduled in a concurrent computation to avoid suspending the client code. Registering interest in the `Read` operation is valid only after the socket is connected, thus the `connected` signal is awaited before registering. Reading is repeated until an end-of-stream is reached, which signals a graceful disconnect by the endpoint. The socket is explicitly closed at the dispatcher and `processRead`'s `doneSignal` is invoked. The actual reading is done in the closure `doRead`, which is invoked by the dispatcher's `waitFor` method. The socket's `readBuffer` is cleared and all available data is read from the NIO channel into the buffer using the socket channel's non-blocking `read` method. If `read` returns `-1` an end-of-stream is signalled, any other value represents the number of bytes successfully read. The contents of the `readBuffer` are copied into an `Array[Byte]` and put into `readCh`, which is concurrently read by client-code.

```
class Socket(implicit val dispatcher: Dispatcher, implicit val
    scheduler: Scheduler) {
  val socketCh: SocketChannel // NIO
```

```
  val readCh = Channel.create[Array[Byte]]
  val readBuffer: ByteBuffer // NIO

  def processRead(): SuspendingAwait = {
    val doneSignal = new Signal
    val wait = dispatcher.waitFor(socketCh) _

    def doRead(ch: SocketChannel): Boolean @suspendable = {
      readBuffer.clear
      val readCnt = ch.read( readBuffer )
      if (readCnt == -1) {
        false
      } else {
        readBuffer.flip
        readCh << Socket.ByteBufferToArray( readBuffer )
        true
    } }
    scheduler execute { reset {
      connected.sawait // suspend
      var EndOfStream = false
      while (!EndOfStream)
        EndOfStream = !doRead(wait(Read))
      dispatcher.close(socketCh)
      doneSignal.invoke
    }}
    doneSignal
} }
```

Listing 5.9: Socket - Processing the read channel

Processing the write channel (`writeCh`) is done in a similiar manner to `processRead`: the processing is scheduled as a concurrent computation to avoid blocking the client and the `connected` signal is awaited before registering interest in the `Write` operation at the dispatcher (see listing 5.10). The write channel's `foreach` method is used to iterate over the bytes put into the channel by client code. Every array of bytes put into the write channel is wrapped in a ByteBuffer and successively processed by the `doWrite` closure. As the socket channel's non-blocking `write` method does not guarantee writing the whole buffer to the socket at once, it is called repeatedly until the buffer is empty. Before each write to the socket, interest in the `Write` operation is registered by using `waitFor`. Finally, when the write channel has been terminated by client code, `foreach` will return and the `doneSignal` is invoked to signal end of writing.

```
class Socket(implicit val dispatcher: Dispatcher, implicit val
   scheduler: Scheduler) {
```

```
  val socketCh: SocketChannel
  val writeCh = Channel.create[Array[Byte]]

  def processWrite(): SuspendingAwait = {
    val doneSignal = new Signal
    val wait = dispatcher.waitFor(socketCh) _

    def doWrite(buf: ByteBuffer): Unit @suspendable = {
      while (buf.remaining > 0) {
        val ch = wait( Write )
        ch.asInstanceOf[SocketChannel].write( buf )
      }
    }

    scheduler execute { reset {
      connected.sawait // suspend
      writeCh.foreach(bytes => doWrite(ByteBuffer.wrap(bytes))
         )
      doneSignal.invoke
    }}
    doneSignal
  }
}
```

Listing 5.10: Socket - Processing the write channel

Reading from and writing to the socket is based on low-level byte arrays. To ease the use of textual protocols such as HTTP, ScalaFlow provides traits that can be mixed in with the `Socket` to enrich its operations. Trait `SocketReadUtils` provides, among others, the method `readStrings`. This method concurrently maps all data from the socket's read channel into a new channel, which contains the read bytes decoded into strings (see listing 5.11).

```
trait SocketReadUtils { self: Socket =>
  def readStrings(implicit charset: Charset): Channel[String] =
    read.map[String](bytes =>
      charset.decode( ByteBuffer.wrap(bytes) ).toString
    )
}
```

Listing 5.11: Trait to enrich `Socket` with a `readStrings` method

5.3. Connection Acceptor

The ability to accept connections on a server socket is provided by the `Acceptor` class (see listing 5.12). By calling `bind`, the underlying server socket is bound to an `InetSocketAddress`. To accept new connections, the method `accept` schedules a concurrent computation that repeatedly registers interest in the `Accept` operation until the acceptor is shut down. When this occurs the returned `doneSignal` is invoked. Accepting a connection is handled in the closure `registerSocket`, which is called by `waitFor` when the selector notices an attempt to connect to the server socket. `registerSocket` is called with the accepted client socket channel as its parameter. The new socket channel is registered at the dispatcher and wrapped in the ScalaFlow `Socket`. The socket's `process` method is executed to process its read and write channel, then finally the socket is appended to the connections channel (`connectionsCh`) which is processed by client code.

```
class Acceptor(implicit val dispatcher: Dispatcher, implicit
    val scheduler: Scheduler) {
  @volatile var _shutdown = false

  val serverSocketCh: ServerSocketChannel
  val connectionsCh = Channel.create[Socket]

  def connections: ChannelTake[Socket] = connectionsCh

  def bind(addr: InetSocketAddress): Unit =
    serverSocketCh.socket.bind( addr )

  def accept(): SuspendingAwait = {
    val doneSignal = new Signal

    dispatcher.register(
      serverSocketCh,
      Some(onClosed),
      Some(onFailure)
    )
    val wait = dispatcher.waitFor( serverSocketCh ) _

    def registerSocket(ch: SocketChannel): Unit @suspendable =
       {
      dispatcher.register(ch, Some(onSocketClosed), Some(
         onSocketFailure))
      val s = new Socket(ch)
      s.process
      connectionsCh << s
```

```
    }

    scheduler execute { reset {
      while (!_shutdown)
        registerSocket( wait(Accept) )
      doneSignal.invoke
    }}

    doneSignal
  }

  def shutdown(): Unit = {
    _shutdown = true
  }

  def onClosed(ch: SelectableChannel): Unit
  def onFailure(failure: Throwable): Unit

  def onSocketClosed(ch: SelectableChannel): Unit
  def onSocketFailure(failure: Throwable): Unit
}
```

Listing 5.12: Acceptor - Accepting connections

5.4. HTTP Processing

ScalaFlow additionally provides a HTTP server capable of processing the HTTP/1.1 protocol [RFC2616]. The protocol specification is implemented using Scala case-classes; the fundamental classes `HttpRequest` and `HttpResponse` are listed in 5.13. The `HttpResponseStatus` implements the `unapply` method and thus can be used as an extractor for pattern matching (see listing 5.14).

```
case class HttpRequest(method: HttpMethod, uri: URI, headers:
   List[HttpHeader], httpVersion:(Int,Int) = (1,1))

case class HttpResponse(status: HttpResponseStatus, headers:
   List[HttpHeader]) {
```

Listing 5.13: `HttpRequest` and `HttpResponse`

```
object HttpResponseStatus {
  val OK = HttpResponseStatus(200, "OK")
  ...
```

```
  sealed trait HttpResponseStatusType
    object Informational extends HttpResponseStatusType {
      def unapply(status: HttpResponseStatus): Boolean = status
          .isInformational
    }
    object Success extends HttpResponseStatusType {
      def unapply(status: HttpResponseStatus): Boolean = status
          .isSuccess
    }
    object Redirection extends HttpResponseStatusType {
      def unapply(status: HttpResponseStatus): Boolean = status
          .isRedirection
    }
    object ClientError extends HttpResponseStatusType {
      def unapply(status: HttpResponseStatus): Boolean = status
          .isClientError
    }
    object ServerError extends HttpResponseStatusType {
      def unapply(status: HttpResponseStatus): Boolean = status
          .isServerError
    }
}

case class HttpResponseStatus(code: Int, phrase: String,
   httpVersion: (Int,Int) = (1,1)) {
  def isInformational = code / 100 == 1
  def isSuccess       = code / 100 == 2
  def isRedirection   = code / 100 == 3
  def isClientError   = code / 100 == 4
  def isServerError   = code / 100 == 5
}
```

Listing 5.14: `HttpResponseStatus`

A `Tokenizer` splits request and response payloads into `Token`s that contain instances of the HTTP specification type classes. The implementation is based on a finite state machine and type classes representing the different states. The tokenizer is initialized with a clean state, applying its `tokenize` method with a raw byte array (`Array[Byte]`), which tokenizes the bytes into a `List[Token]`, which is returned. Multiple invocations of `tokenize` continue processing based on the tokenizer's internal state, which can be `reset`. This design directly feeds the `Socket`'s `read` channel bytes into the `tokenizer` method.

```
object Tokenizer {
  sealed trait State
  object State {
    case object Init extends State
```

```
    ...
    case object InitChunk extends State
    case class Chunk(size: Int) extends State
    case class RemoveChunkBoundary(nextState: State) extends
       State
    ...
    case object Success extends State
    case class Failure(cause: String) extends State
  }

  sealed trait Token
    case class Request(request: HttpRequest) extends Token
    case class Status(status: HttpResponseStatus) extends Token
    ...
          case object StartChunks extends Token
          case class Chunk(bytes: Array[Byte]) extends Token
          case object EndChunks extends Token
}

class Tokenizer {
  var state: State = State.Init
  def tokenize(bs: Array[Byte]): List[Token]
}
```

Listing 5.15: HTTP Tokenizer

The `HttpProcessor` class is bound to a `Socket` and produces `HttpRequest`s or `HttpResponse`s based on the socket's `read` channel. Additionally, it is able to issue a `HttpRequest` to the `Socket`'s `write` channel.

The processor uses a *Pipeline* design to concurrently transform the raw bytes from the socket into requests and responses. The first pipeline stage consumes bytes from the socket's `read` channel, tokenizes them using the `Tokenizer` and puts the `Token`s into a `Channel[Token]`. The second stage consumes the `Channel[Token]` and transforms the `Token`s to `HttpRequest`s or `HttpResponse`s.

The tokenizing stage is implemented by applying the `Channel`'s higher-order function `flatMapIter` with a transformation function to the `Socket`'s `read` channel. The transformation function is of type `Array[Byte] => List[Token]`, where `List` is a subtype of `Iterable`. This function, concurrently executed by `flatMapIter`, feeds each `Array[Byte]` into the `Tokenizer`'s `tokenize` function which produces a `List[Token]`. `flatMapIter` appends this list element by element to the `Channel[Token]`. Therefore the first stage can be defined as `val tokens = socket.read.flatMapIter(bytes => tokenizer.tokenize(bytes)`

There are two alternative second stages: one produces `HttpRequest`s, the other `HttpResponse`s. Their implementations only differ in details which are omitted from this section. To achieve concurrent execution, the second stage also uses a higher-order function of the `Channel` class, which concurrently executes the passed function. The function transforming the tokens has type `Token => Option[HttpRequest]`. It aggregates `Token`s and yields `Some[HttpRequest]` if the collected tokens form a valid `HttpRequest`, or `None` if further `Token`s are required. Thus the transformation function is passed to the higher-order function `collect` which accepts functions of type `A => Option[B]`.

The pipeline stages are created on demand using `lazy val`s to store the channels. The first `lazy val` stores the `Channel[Token]`, the others store a `Channel[HttpRequest]` and a `Channel[HttpResponse]` respectively. The processor provides the methods `requests` and `responses` which return the `Channel[HttpRequest]` and `Channel[HttpResponse]` respectively. When accessing the lazy references to the channels, the lazy-val mechanism creates their instances and thus starts the pipeline processing. However client code must not invoke *both* the `requests` and `responses` methods, as they compete for the single `Channel[Token]`, thus clients can either use the processor for processing requests or for processing responses.

```
class HttpProcessor(val socket: Socket, val forceCharset:
   Option[Charset] = None) {

  // issue a http-request by writing the request's raw bytes to
      the socket
  def request(req: HttpRequest): Unit @suspendable =
    socket.write << Socket.ByteBufferToArray( req.toBytes )

  def requests: ChannelTake[HttpRequest]   = chRequests
  def responses: ChannelTake[HttpResponse] = chResponses

  // lazy: created on first usage,
  // triggered by requests- or responses-method
  lazy val chTokens: Channel[Tokenizer.Token] = {
    val t = new Tokenizer

    // mapping the bytes to a List[Token] (Iterable),
    // flat-append the list to the new channel created by
       flatMapIter
    socket.read flatMapIter { bytes => {
      try   { t.tokenize(bytes)       }
      catch { case e:Exception => Nil }
    }}
  }
```

```
  // lazy: created on first usage, triggered by requests-method
  lazy val chRequests: Channel[HttpRequest] = {
    def create(t: Token): Option[HttpRequest] = {
      // aggregating tokens into a HttpRequest instance
    }

    // catching any exceptions while creating HttpRequests
    def tryCreate(t: Token): Option[HttpRequest] =
      try   { create( t )              }
      catch { case e:Exception => None }

    // iterating over tokens-channel
    chTokens collect (tryCreate _)
  }

  // lazy: created on first usage, triggered by responses-
     method
  lazy val chResponses: Channel[HttpResponse] = {
        def create(t: Token): Option[HttpRequest] = t match {
      // aggregating tokens into a HttpResponse instance
    }

    // catching any exceptions while creating HttpResponses
    def tryCreate(t: Token): Option[HttpRequest] =
      try   { create( t )              }
      catch { case e:Exception => None }

        // iterating over tokens-channel
    chTokens collect (tryCreate _)
  }
}
```

Listing 5.16: HTTP Processor

5.5. Conclusion

Concluding the design of ScalaFlow's networking components, a class diagram is presented in 5.1. The `Socket` provides read and write access to the underlying NIO-socket through `Channel`s which transport raw byte-arrays. The `Acceptor` uses a `Channel` to enqueue client-connections represented by `Socket` instances. The class diagram of the HTTP-processing is depicted in figure 5.2. The central component `HttpProcessor` transforms the `Socket`'s read-channel to instances of the HTTP-

protocol case classes. The processor uses the `Tokenizer` to preprocess the byte sequences from the `Socket`'s `read` channel. The tokens are then aggregated and enqueued into the `requests` or `responses` `Channel`s.

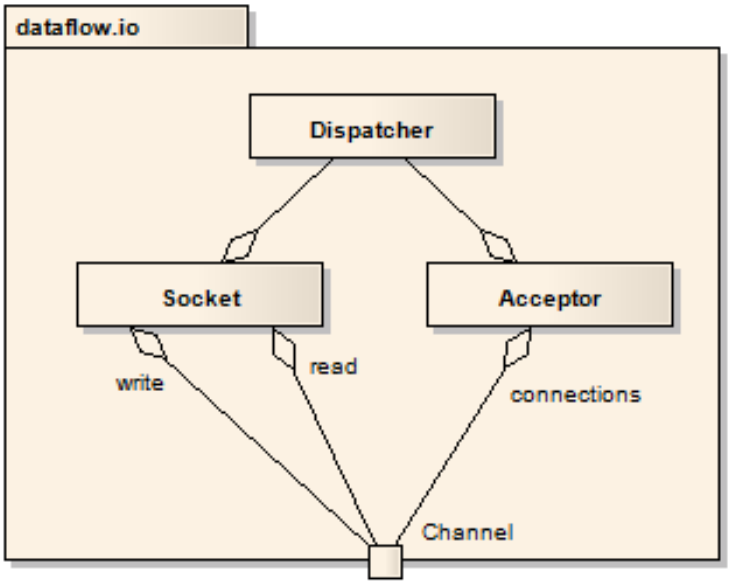

Figure 5.1.: Class diagram of ScalaFlow's networking components

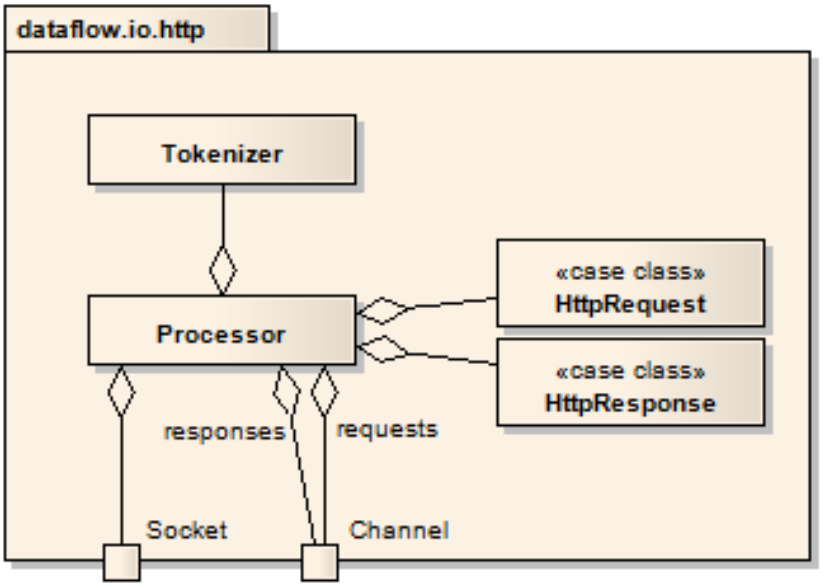

Figure 5.2.: Class diagram of ScalaFlow's HTTP processing

6

ScalaFlow in Action

This chapter illustrates the intended use of the ScalaFlow framework. The first section 6.1 demonstrates explicit usage of ScalaFlow in conjunction with a dataflow variable, emphasizing on types and avoiding any shortcuts. The next section 6.2 contains examples using `Variable`s with ScalaFlow's terse DSL notation, followed by section 6.3 which presents dataflow `Channel`s. The networking components `Socket` and `Acceptor` are demonstrated in section 6.4. Finally a basic web server is developed and discussed.

6.1. Explicit Usage

Listing 6.1 demonstrates explicit usage of the ScalaFlow framework. This listing avoids type-inference and the terse syntax through implicit variables and operator-aliases. DSL operations such as `flow` are not used in favour of explicit `Flow` instantiation. Important behavior is denoted by comments throughout the listing.

```
object Explicit {
  val scheduler: Scheduler = new ThreadPoolScheduler
```

```
  def main(args: Array[String]): Unit = {
    val v: Variable[Int] = new Variable[Int]()(scheduler)

    // computations
    val c1: (() => Int  @dataflow) = () => v.get
    val c2: (() => Unit @dataflow) = () => v.set(42)

    // flows
    val f1: FlowResult[Int]  = new Flow[Int](c1)(scheduler).
       execute.result
    val f2: FlowResult[Unit] = new Flow[Unit](c2)(scheduler).
       execute.result

    // results
    val r1: Int  = f1.get // blocks
    val r2: Unit = f2.get // blocks
    println("r1="+r1+" r2="+r2) // r1=42 r2=()

    scheduler.shutdown.await
  }
}
```

Listing 6.1: Explicit usage of ScalaFlow

6.2. Using Variables

Listing 6.2 demonstrates how to use `Variable` with ScalaFlow's terse DSL notation. In contrast to listing 6.1, the `scheduler` is created as an implicit reference and hence is implicitly passed to the `Variable` and `Flow`. Instead of creating a `ThreadPoolScheduler`, this example uses a `DaemonThreadPoolScheduler` which is automatically disposed when the program terminates, avoiding the need to explicitly shut it down. Furthermore the `Variable`'s DSL aliases are used: `get` is replaced by an application and `set` is replaced by the `:=` alias.

```
object Var1 {
  implicit val scheduler = new DaemonThreadPoolScheduler
  def main(args: Array[String]): Unit = {
    val v = new Variable[Int]
    val f1 = flow { v()     }
    val f2 = flow { v := 42 }
    println("r1="+f1.get+" r2="+f2.get)
} }
```

Listing 6.2: Using a Variable in terse DSL notation

6.3. Using Channels

Similiar to the previous section, listing 6.3 uses ScalaFlow's terse DSL notation to present the intended usage of channels. Listing 6.3 appends two values to a channel. The channel is created by ScalaFlow's default factory which creates bounded channels with a default capacity of 1000.

```
object Channel1 {
  implicit val scheduler = new DaemonThreadPoolScheduler
  def main(args: Array[String]): Unit ={
    val ch = Channel.create[Int]

    // consumer-flow
    val f = flow {
      val v1 = ch()
      val v2 = ch()
      v1 + v2
    }
    // producer-flow
    flow {
      ch << 11
      ch << 31
    }
    println("result: " + f.get)
} }
```

Listing 6.3: Using a Channel in terse DSL notation

Listing 6.4 introduces an additional consumer that competes with the first one. They are scheduled nondeterministically, so the ordering of their take operations and hence the program's outcome will vary between runs. The channel is again bounded by a default capacity and is not terminated.

```
object Channel2 {
  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit ={
    val ch = Channel.create[Int]

    // consumer-flow #1
    val f1 = flow { ch() + ch() }

    // consumer-flow #2
    val f2 = flow { ch() }

    // producer-flow
```

```
    flow {
      ch << 1
      ch << 2
      ch << 3
    }

    val r1 = f1.get
    val r2 = f2.get
    println(r1 + " - " + r2 + " = " + (r1-r2))
  }
}
```

Listing 6.4: Multiple consumers taking from a channel

Listing 6.5 demonstrates program behavior when terminating a channel. Consumers that read a terminated channel have to deal with the resulting `TerminatedChannel` exception. The listing terminates with `result: 0` since the second application of `safeTake` has to deal with the exception.

```
object Channel3 {
  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit ={
    val ch = Channel.create[Int]

    def safeTake: Int @dataflow =
      try { ch() }
      catch { case TerminatedChannel => -1 }

    // consumer-flow
    val f = flow {
      val v1 = safeTake
      val v2 = safeTake // raises TerminatedChannel
      v1 + v2
    }

    // producer-flow
    flow {
      ch << 1
      ch <<# // terminate
    }

    println("result: " + f.get)
  }
}
```

Listing 6.5: Consumer reading a terminated channel

Using Scala's for-comprehension on a `Channel` is demonstrated in listing 6.6. The for-loop terminates if the channel is terminated. Not terminating the channel results in the for-loop being suspended while waiting for further values. The for-loop is expanded into the application of `Channel.foreach`. Finally the program `await`s the first flow, which blocks the main thread until the first flow has finished executing.

```
object Channel4 {
  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit ={
    val ch = Channel.create[Int]

    // consumer-flow
    val f = flow {
      for (x <- ch)
        println(x)
    }

    // producer-flow
    flow {
      ch << 1
      ch << 2
      ch << 3
      ch <<# // terminate
    }

    f.await
  }
}
```

Listing 6.6: For-comprehension on a channel, using `foreach`

Listening 6.7 uses the for-comprehension expanded into applications of `map` and `filter`. The second flow concurrently maps values from `xs` into the new channel `ys`, thus returning a `FlowResult[Channel[Int]]`. The third flow uses `sget` (suspending get) on the `FlowResult` to get the `ys` (applying `get` is a thread-blocking operation which is to be avoided in `flow` environments). That said it is more idiomatic to use a `Variable[Channel[Int]]` to communicate the `ys` between the flows.

```
object Channel5 {
  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit ={
    val xs = Channel.create[Int]

    // producing xs
```

```
    flow {
      xs << 1
      xs << 2
      xs << 3
      xs <<# // terminate
    }

    // consuming xs, producing ys
    val f1 = flow {
      val ys = for (x <- xs; x > 1; y = x*x)
        yield y
    }

    // consuming ys
    val f2 = flow {
      val ys = f1.sget // suspending get
      for (y <- ys)
        println(y)
    }

    f2.await
  }
}
```

Listing 6.7: Concurrent mapping and waiting for another flow's result

The `Channel` class provides multiple factory methods, to produce channels with different types of flow-control behavior (see listing 6.8). The listing also demonstrates the implicit conversion of Scala's `Iterable` to ScalaFlow's `DataFlowIterable`, which is CPS-aware. The conversion is forced by using the identity operation `dataflow` on the `Iterable`.

```
object Channel6 {
  import DataFlowIterable.convert

  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit ={

    val chEager   = Channel.createEager[Int]
    val chLazy    = Channel.createLazy[Int]
    val chBounded = Channel.createBounded[Int](3)

    // eager producer, no suspending
    // producing 1..10
    val f1 = flow { for (x <- (1 to 10).dataflow) {
      println("eager: " + x)
      chEager << x
```

```
    }}

    // lazy producer, suspended by default,
    // resumed on consumer's demand.
    // producing 1..10
    val f2 = flow { for (x <- (1 to 10).dataflow) {
      println("lazy: " + x)
      chLazy << x
    }}

    // bounded producer, suspended when capacity reached
    // producing 1..10
    val f3 = flow { for (x <- (1 to 10).dataflow) {
      println("bounded: " + x)
      chBounded << x
    }}

    flow {
      // not consuming eager-channel at all
      // chEager()

      chLazy()

      chBounded()
    }

    f1.await && f2.await && f3.await
  }
}
```

Listing 6.8: Demonstrating Flow-Control and the CPS-aware Iterable-Wrapper

The final listing 6.9 demonstrates usage of multiple higher-order functions on channels. However the selection of demonstrated functions is not exhaustive. The flows communicate via dataflow variables and all higher-order functions work concurrently. The termination of the first channel is automatically propagated to all following channels that are created by higher-order functions, thus terminating the enclosing program.

```
object Channel7 {
  import DataFlowIterable.convert

  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit ={

    val ch = Channel.create[Int]
```

```
// cps-aware sum
def cpsSum(x: Double, y: Double): Double @dataflow =
  x + y

// producing 1..100, terminating
flow {
  for (i <- (1 to 100).dataflow) {
    ch << i
  }
  ch <<#
}

val collected = new Variable[Channel[Double]]
flow {
  // combination of map and filter
  collected := ch.collect(i =>
    if (i % 2 == 0) None // skip evens
    else            Some( i.toDouble ) // transform odds
  )
}

val split1, split2 = new Variable[Channel[Double]]
flow {
  // splitting into two channels based on predicate
  val (s1,s2) = collected().split(i => i < 50)
  split1 := s1 // <  50
  split2 := s2 // >= 50
}

val dup1, dup2 = new Variable[Channel[Double]]
flow {
  // duplicating
  val (d1,d2) = split1().duplicate
  dup1 := d1 // < 50
  dup2 := d2 // < 50
}

val squares = new Variable[Channel[Double]]
flow {
  // another way to calculate the squares...
  val d1  = dup1()
  val d2  = dup2()
  val sqs = for (x <- d1; y <- d2) yield x*y
  squares := sqs
}

val squareSum = new Variable[Double]
flow {
  // reducing to sum
```

```
      squareSum := squares().reduce(cpsSum _)
    }

    val sum = new Variable[Double]
    flow {
      // folding to sum
      sum := split2().fold(0.0)(cpsSum _)
    }

    val f = flow {
      println("square-sum: " + squareSum())
      println("sum: " + sum())
    }

    f.await
  }
}
```

Listing 6.9: Multiple higher-order functions on `Channel`s

6.4. Networking

This section demonstrates the intended use of the `Dispatcher`, `Socket`, `Acceptor` and the HTTP processing components. The first example shows how to communicate with a host using the `Dispatcher` and `Socket`. The HTTP protocol is used to demonstrate connecting, writing requests as raw bytes and processing response bytes. The second example creates a simple Echo server [Pos83] that accepts connections from clients and echos back all incoming data. Finally the third example describes a simple web server demonstrating ScalaFlow's HTTP components, which are built using the `Dispatcher`, `Socket` and `Acceptor` classes.

The first example in listing 6.10 connects to a host addressed by a `java.net.InetSocketAddress`. The socket's `connect` method suspends the first flow until the connection is successfully established. When resumed, the flow invokes `process` which concurrently processes the socket's read and write channels. The flow `sawait`s the `SuspendingAwait` that is returned by `process`. It is invoked when both the read channel and the write channel are terminated and the socket is closed. The `read` channel is terminated when the socket encounters *EndOfStream* when reading from the remote host. The `write` channel has to be explicitly terminated.

The second flow simply `sawait`s the socket's `connected` signal, to show the intended use of the socket's signals. The third flow writes a raw byte request to the socket's `write` channel. Since the socket's internal processing of its `read` and `write` channels is synchronized on the `connected` signal, reading from and writing to the socket channels is allowed at any time.

The fourth flow concurrently decodes the bytes from the socket's `read` channel into a `Channel[Array[Char]]`. The flow passes this new channel to the fifth flow via the `chars` variable. The fifth and final flow uses the for-comprehension to iterate over the `Channel[Array[Char]]`, which is bound to variable `chars` by the fourth flow. The program `await`s the `FlowResult`s of flows one and five, since they are the last flows to return.

```
object IO1 {

  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit = {
    implicit val dispatcher = Dispatcher.start
    val sock = new Socket
    val heise = new InetSocketAddress("www.heise.de", 80)

    val f1 = flow {
      sock.connect(heise)
      println("connected to heise")
      sock.process.sawait
      println("socket processed, no more data to write & server
          disconnected")
    }

    val f2 = flow {
      sock.connected.sawait
      println("signal: connected to heise")
    }

    val f3 = flow {
      sock.write << "GET http://www.heise.de/ HTTP/1.1\r\nHost:
          www.heise.de\r\n\r\n".getBytes("ASCII")
      sock.write <<#;
    }

    val cs = java.nio.charset.Charset.forName("UTF-8")
    val chars = new Variable[ Channel[Array[Char]] ]

    val f4 = flow {
```

```
      def bytesToChars(bs: Array[Byte]): Array[Char] @dataflow
          =
        cs.decode( java.nio.ByteBuffer.wrap(bs) ).array

      chars := sock.read.map(bytesToChars _)
    }

    val f5 = flow {
      for (cs <- chars.get) {
        println(cs.length + " bytes")
        //print( cs.mkString("") )
      }
    }

    f1.await && f5.await
  }
}
```

Listing 6.10: Connecting to a host using the `Dispatcher` and `Socket`

The second example in listing 6.11 accepts client connections on a specified IP address by invoking the `Acceptor`'s `accept` method with a local `java.net.InetSocketAddress` in the first flow. The method returns a `SuspendingAwait` and concurrently accepts connections. The returned `SuspendingAwait` is not awaited in this example.

The second flow uses the for-comprehension on the acceptor's `connections` channel. The channel contains client sockets that encapsulate the accepted connection requests. The sockets are already in processing mode since the `Acceptor` internally invoked the socket's `process` method. For each client socket from the `connections` channel, a new embedded flow is created to process the request concurrently. The embedded flows are executed independently and use the for-comprehension to process their socket's `read` channel. Each `Array[Byte]` from the `read` channel is written back to the socket's `write` channel, thus the example is echoing back each client's input.

```
object IO2 {

  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit = {
    implicit val dispatcher = Dispatcher.start
    val ator = new Acceptor
    val addr = new InetSocketAddress("localhost", 1337)

    val f1 = flow {
      ator accept addr
    }
```

```
    val f2 = flow {
      for (sock <- ator.connections) { flow {
        for (bytes <- sock.read) {
          sock.write << bytes
        }
      }}
    }

    f2.await
  }
}
```

Listing 6.11: Echo-Server using the `Dispatcher` and `Acceptor`

The final example uses ScalaFlow's HTTP processing components to create a simple web server, which accepts HTTP connections and responds with a HTML page that summarizes protocol information (see listing 6.12). Similiar to the previous example, the acceptor is bound to a local `InetSocketAddress`, with the first flow accepting client connections by invoking its `accept` method. As before the second flow creates a new embedded flow for each client connection. Each client flow creates a `HttpProcessor` that processes the client socket. A for-comprehension is used to iterate over the `requests`, which are concurrently produced by the HTTP processor based on the socket's raw input bytes. Each request is an instance of `HttpRequest` which is ScalaFlow's type-safe HTTP representation. `createResponse` creates a `HttpResponse` using Scala's XML literals. The response is written to the client socket's `write` channel and displayed in the client's browser.

```
object IO3 {
  implicit val scheduler = new DaemonThreadPoolScheduler

  def main(args: Array[String]): Unit = {
    implicit val dispatcher = Dispatcher.start
    val ator = new Acceptor
    val addr = new InetSocketAddress("localhost", 8080)

    val f1 = flow {
      ator accept addr
    }

    val f2 = flow {
      for (socket <- ator.connections) { flow {
        val http = new HttpProcessor(socket)
        for (request <- http.requests) {
          val resp = createResponse(request)
```

```
          socket.write << Socket.ByteBufferToArray(resp.toBytes
             )
        }
      }}
    }

    f2.await
  }

  def createResponse(request: HttpRequest): HttpResponse = {
    val headers = List[HttpHeader](
      HttpContentType.HTML.toHeader("UTF-8"),
      HttpHeader("X-HTTPD", "ScalaFlow2010")
    )
    val response = HttpResponse(HttpResponseStatus.OK, headers)

    val res = <html>
<head><title>ScalaFlow</title></head>
<body>
<div>
  <h2>Request-URI</h2>
  <span>{ request.uri.toString }</span>
</div>
<div>
  <h2>Request-Headers</h2>
  <ul>
  { request.headers.map(h => {
    <li>
      <span><strong>{ h.name }</strong></span>
      <span>{ h.value }</span>
    </li>
  }) }
  </ul>
</div>
<div>
  <h2>Response-Headers</h2>
  <ul>
  { headers.map(h => {
    <li>
      <span><strong>{ h.name }</strong></span>
      <span>{ h.value }</span>
    </li>
  }) }
  </ul>
</div>
<div>
  <h2>TimeStamp</h2>
  <span>{ System.currentTimeMillis }</span>
</div>
```

```
</body>
</html>

    response.addBody("""<!DOCTYPE html PUBLIC "-//W3C//DTD
       XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/
       xhtml1-strict.dtd">""")
    response.addBody( res.toString )

    response
  }
}
```

Listing 6.12: WebServer

7 Conclusion

To conclude this thesis, this chapter begins with a summary of the achieved results. Subsequently the framework is critically assessed and recommendations for future work are made.

7.1. Summary

This thesis has described the design and implemention of a framework for creating concurrent programs with DataFlow execution. The framework, named ScalaFlow, allows creation of embedded DataFlow environments within Scala programs. The DataFlow Model is based on the *Data Driven Concurrency Model*, extended with nondeterminism. ScalaFlow features dataflow *Variables*, *Signals*, communication *Channels*, *Pipelines* and networking components.

ScalaFlow uses a type-driven, delimited transformation into *Continuation-Passing-Style* (CPS) which enables capturing of continuations. The framework's aim is to avoid thread-blocking operations to achieve scalability. By capturing continuations, the framework transparently suspends and resumes computations without blocking any

thread. Computations are resumed by submitting their continuation to a Scheduler. ScalaFlow provides three *Schedulers*: the first utilizes a fixed pool of threads. The second scheduler is a variation of the first one, creating daemon-threads that are automatically disposed when the program terminates. The third scheduler uses the efficient Fork/Join framework, which is in development for Java7.

Dataflow *Variables* are single-assignment and suspend their readers until the variable is bound. *Signals* suspend their listeners until the signal is invoked. In contrast to variables, signals do not communicate any value. *Channels* are streams of dataflow variables used to communicate between concurrent dataflow computations. Consumers of empty channels are suspended until the channel is filled. Additionally, channels suspend and resume producers based on the channel's capacity; an eager channel does not suspend producers. A lazy channel suspends producers by default and resumes them based on consumer demand. Any capacity in between constitutes a bounded buffer that suspends producers when the channel is filled up to its capacity. A bounded channel resumes producers in first-in, first-out (FIFO) order when consumers drain the channel. The *Pipeline* is a lightweight interface that makes use of the channel's concurrent higher-order function `map` to connect channels to form a pipeline, where each mapping function represents a concurrent pipeline stage.

ScalaFlow additionally features networking components to connect to remote systems, create servers or build up a distributed system. The *Dispatcher* is a lightweight event dispatcher for NIO readiness events. The CPS transformation is used to register continuations as event handlers which counteracts inversion of control and allows creation of NIO applications with an uncluttered logical flow. The *Socket* represents a bidirectional connection endpoint and provides read and write dataflow channels for network communication. The *Acceptor* represents a server socket that accepts incoming connection requests. It provides a dataflow channel to pass sockets representing the client-connections to handler flows. The *HTTP Processor* transforms a socket's read and write channels, which only transport raw byte arrays, to instances of ScalaFlow's HTTP interface.

In conclusion, the framework provides embedded dataflow environments, enabling creation of concurrent programs that benefit from implicit synchronization, data-driven dynamic dependencies between the computations and declarative single-assignment variables. Furthermore, the framework provides basic networking components sufficient to create simple distributed systems and communicate via HTTP out-of-the-box.

Based on the data-driven DataFlow Model, the semantic distance of programs is reduced.

7.2. Assessment

ScalaFlow's suitability for practical applications is demonstrated in the examples section and the projects listed in the appendix. ScalaFlow combines object-oriented programming, used to implement the framework's foundation, with functional programming, which allows use of ScalaFlow via a terse DSL notation with concurrent higher-order functions.

The foundation of the dataflow model is the dataflow *Variable*, which uses a lock-free implemented based on atomic references, allowing it to perform well under consumer contention. By contrast *Channel*s uses heavy locking to manage their internal streams and to provide flow-control to both consumers and producers. Future work should attempt to design a more efficient locking scheme or even a lock-free *Channel* implementation. Although performance profiling and evaluation of the framework was not undertaken in this thesis, tests and microbenchmarks indicate reasonable performance. However further work should profile the channel's behavior under heavy load with fixed capacity and multiple consumers and producers.

Using the for-comprehension on Scala Collections is not possible in CPS-transformed code. The for-comprehension is expanded into applications of `map` and `foreach` on the collections which lack the `@cps[A]` annotation. Therefore, statements like `flow { for (i <- 1 to 10) channel put i }` cannot be CPS-transformed, causing a compilation failure. ScalaFlow provides an implicit conversion from Scala's `Iterable` to the CPS-aware `DataFlowIterable` wrapper by using the `dataflow` keyword on the `Iterable`: `flow { for (i <- (1 to 10).dataflow) channel put i }`. However the wrapper only implements `foreach` and `map`; this mechanism should to be heavily extended in future work.

Undertaking network communication imposes numerous potential sources of failure on a program: from establishing the connection to sending or receiving data, each operation has various failure modes which need to be handled appropriately. ScalaFlow's networking library only provides very basic failure handling: it is not ready for production use. Future work should improve its failure handling capability and assess its use of Java's NIO framework. ScalaFlow might benefit from switching

to NIO2, which provides non-blocking operations on files and is in development for Java 7.

ScalaFlow and the projects listed in the appendix are public, BSD licensed projects on github. It is expected that some of the stated problems will be improved or solved in future development cycles. Support from the Scala community is appreciated.

A

Appendix Source Code and Projects

The full source code of ScalaFlow can be found at `http://github.com/hotzen/-ScalaFlow` and is available under the BSD license. ScalaFlow includes the packages `dataflow`, `dataflow.io`, `dataflow.io.http` and `dataflow.util`. Additionally, package `dataflow.tests` contains many more examples than presented in the examples chapter.

Building on ScalaFlow, there are two projects hosted at github. The first, called *MetaFlow* is available at `http://github.com/hotzen/MetaFlow` and combines heuristics with ScalaFlow's `Sockets` and HTTP processing to create a web crawler that extracts informations about movies. It is still work in progress and far from completion but can be studied for a real-word example on using ScalaFlow.

The second project is called *CompFlow* and is available at `http://github.com/-hotzen/CompFlow`. Based on ScalaFlow, it suspends computations, sends them over the network using ScalaFlow's `Acceptor` and `Socket` and resumes the computation on remote nodes. The project is inspired by Swarm [Cla] and not yet working, since Java's serializing mechanism still raises problems in combination with ScalaFlow.

B

Installation Guide

A CD is attached to this thesis, containing a PDF version of the thesis as well as the source code of the developed ScalaFlow framework. Additionally, the framework is provided as a self-contained JAR-archive. ScalaFlow is also publicly available under `www.github.com/hotzen/ScalaFlow`.

The source code contains a project definition for the simple-build-tool (sbt) [H+a] and can be built by invoking `sbt compile` inside the `ScalaFlow` directory. The directory-structure follows the recommendations of the SBT-project [H+b]. A project definition for the Eclipse IDE can be created by invoking `sbt eclipse`.

The framework is implemented in the following packages: `dataflow`, `dataflow.io` and `dataflow.io.http`. The basic framework is imported by the single statement: `import dataflow._`. To use the implicit conversion of Scala Collections to the CPS-aware `Iterable`-wrapper, an additional `import DataFlowIterable._` is required though. The networking components have to be explicitly imported from package `dataflow.io`, too. All examples listed in this thesis are available in package `dataflow.tests`.

Bibliography

[AVWW96] Armstrong, Joe ; Virding, Robert ; Wikstroem, Claes ; Williams, Mike: *Concurrent Programming in Erlang*. Prentice-Hall, 1996

[BCLGH93] Benveniste, Albert ; Caspi, Pauls ; Le Guernic, Paul ; Halbwachs, Nicolas: *Data-flow Synchronous Languages*. 1993

[Bon09] Boner, Jonas: *Scala Dataflow*. http://github.com/jboner/scala-dataflow. Version: 2009, last checked: 2010-09-01

[Cla] Clarke, Ian: *swarm-dpl - A transparently scalable distributed programming language*. http://code.google.com/p/swarm-dpl/, last checked: 2010-09-01

[Coma] Community, Haskell: *HaskellWiki - Monad*. http://www.haskell.org/haskellwiki/Monad#Special_notation, last checked: 2010-09-01

[Comb] Community, Java: *RFE - Reification of generic type parameters*. http://bugs.sun.com/view_bug.do?bug_id=5098163, last checked: 2010-09-01

[Comc] Community, Scheme: *Scheme Wiki - Call with current continuation*. http://community.schemewiki.org/?call-with-current-continuation, last checked: 2010-09-01

[Dou09] Dougherty, Rich: *Tail calls, @tailrec and trampolines*. http://blog.richdougherty.com/2009/04/tail-calls-tailrec-and-trampolines.html. Version: 2009, last checked: 2010-09-01

[Ell09] Elliott, Conal: Push-pull functional reactive programming. In: *Haskell Symposium 2009*, 2009 http://conal.net/papers/push-pull-frp/

[Eva04] Evans, Eric: *Domain-Driven Design*. Addison-Wesley, 2004

[Far08] Farnham, Kevin: *Interesting Multicore Crisis Graph and Analysis*. http://software.intel.com/en-us/blogs/2008/01/17/interesting-multicore-crisis-graph-and-analysis/. Version: 2008, last checked: 2010-09-01

[FE] Fowler, Marting ; Evans, Eric: *Fluent Interface*. http://martinfowler.com/bliki/FluentInterface.html, last checked: 2010-08-04

[Fow05] Fowler, Martin: *Inversion of Control*. http://martinfowler.com/bliki/InversionOfControl.html. Version: 2005, last checked: 2010-08-10

[GBB+06] Goetz, Brian ; Bloch, Joshua ; Bowbeer, Joseph ; Lea, Doug ; Holmes, David ; Peierls, Tim: *Java Concurrency in Practice*. Addison-Wesley, 2006

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns - Elements of Resusable Object-Oriented Software*. Addison-Wesley, 1995

[H+a] Harrah, Mark et al.: *simple-build-tool - A build tool for Scala*. http://code.google.com/p/simple-build-tool/, last checked: 2010-09-01

[H+b] Harrah, Mark et al.: *simple-build-tool - Setup*. http://code.google.com/p/simple-build-tool/wiki/Setup, last checked: 2010-09-01

[HO07] Haller, Philipp ; Odersky, Martin: Actors that Unify Threads and Events / Programming Methods Laboratory, Ecole Polytechnique Federale de Lausanne. Version: 2007. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.5844&rep=rep1&type=pdf, last checked: 2010-08-04. 2007. – Forschungsbericht

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice Hall International, 1985

[Lea00] Lea, Doug: A Java fork/join framework. In: *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, Department of Computer Science, University of Rochester, 2000 http://gee.cs.oswego.edu/dl/papers/fj.pdf

[Mai10] Maier, Ingo: *scala.react*. http://lamp.epfl.ch/~imaier/. Version: 2010, last checked: 2010-09-01

[Moo65] Moore, Gordon E.: *Cramming more components onto integrated circuits*. ftp://download.intel.com/research/silicon/moorespaper.pdf. Version: 1965, last checked: 2010-09-01

[MRO10] Maier, Ingo ; Rompf, Tiark ; Odersky, Martin: Deprecating the Observer Pattern / Programming Methods Laboratory, Ecole Polytechnique Federale de Lausanne. Version: 2010. http://infoscience.epfl.ch/record/148043/files/DeprecatingObserversTR2010.pdf, last checked: 2010-08-04. 2010. – Forschungsbericht

[MS96] Michael, Maged M. ; Scott, Michael L.: Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In: *Annual ACM Symposium on Principles of Distributed Computing*, Department of Computer Science, University of Rochester, 1996 http://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf

[MSM09] Mattson, Timothy G. ; Sanders, Beverly ; Massingill, Berna: *Patterns For Parallel Programming*. Addison-Wesley, 2009

[O+] Odersky, Martin et al.: *Scala Reference Manuals*, http://www.scala-lang.org/node/198, last checked: 2010-08-04

[Ode08] Odersky, Martin: *Tail calls via trampolining and an explicit instruction*. http://scala-programming-language.1934581.n4.nabble.com/Tail-calls-via-trampolining-and-an-explicit-instruction-td2007474.html#a2007485. Version: 2008, last checked: 2010-09-01

[Ode10] Odersky, Martin: *The Scala Language Specification*. http://www.scala-lang.org/docu/files/ScalaReference.pdf. Version: 2010, last checked: 2010-09-01

[OSV08] Odersky, Martin ; Spoon, Lex ; Venners, Bill: *Programming in Scala*. Artima, 2008

[Pos83] Postel, J.: *Echo Protocol*. RFC 862 (Standard). http://www.ietf.org/rfc/rfc862.txt. Version: May 1983 (Request for Comments)

[Rep10a] Repository, Portland P.: *Dead Lock*. http://c2.com/cgi/wiki?DeadLock. Version: 2010, last checked: 2010-09-01

[Rep10b] Repository, Portland P.: *Race Condition*. http://c2.com/cgi/wiki?RaceCondition. Version: 2010, last checked: 2010-09-01

[Rep10c] Repository, Portland P.: *Race Condition*. http://c2.com/cgi/wiki?FutureValue. Version: 2010, last checked: 2010-09-01

[RFC2616] Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). http://www.ietf.org/rfc/rfc2616.txt. Version: June 1999 (Request for Comments). – Updated by RFC 2817

[RH04] Roy, Peter van ; Haridi, Seif: *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004

[RMO09] Rompf, Tiark ; Maier, Ingo ; Odersky, Martin: Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform / Programming Methods Laboratory, Ecole Polytechnique Federale de Lausanne. Version: 2009. http://lamp.epfl.ch/~rompf/continuations-icfp09.pdf, last checked: 2010-08-04. 2009. – Forschungsbericht

[Rot07] Roth, Gregor: *Architecture of a Highly Scalable NIO-Based Server*. http://today.java.net/article/2007/02/08/architecture-highly-scalable-nio-based-server. Version: 2007, last checked: 2010-09-01

[Sch94] Schmidt, Douglas C.: *Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*. http://www.cs.wustl.edu/~schmidt/PDF/PLoP-94.pdf. Version: 1994, last checked: 2010-09-01

[War07] Warfield, Bob: *A Picture of the Multicore Crisis*. http://smoothspan.wordpress.com/2007/09/06/a-picture-of-the-multicore-crisis/. Version: 2007, last checked: 2010-09-01

Affidavit

I hereby declare that this thesis has been written independently by me, solely based on the specified literature and resources. All ideas that have been adopted directly or indirectly from other works are denoted appropriately. This thesis has not been submitted to any other board of examiners in its present or a similar form and was not yet published in any other way.

Wedel, September 2nd, 2010

Kai H. Meder