Behaviour-Interaction-Priority in Functional Programming Languages

Formalisation and Implementation of Concurrency Frameworks in Haskell and Scala

Romain Edelmann

This thesis was done under the supervision of
Prof. Joseph Sifakis
and
Dr. Simon Bliudze

RiSD laboratory - IC section
École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
January 2015

Acknowledgements

First of all, I would like to thank my family. Without your endless support and care, this work would not have seen the light of day. Thank you for providing me with a very supportive environment!

I would also like to thank Prof. Sifakis and Dr. Bliudze, with whom I had very insightful discussions. I would also like to thank them for giving me the opportunity to take part in the Research Scholar program at EPFL and work at the RiSD laboratory prior to this thesis. This whole experience has been very instructive to me. Obviously, this work would not have been possible without this great opportunity that was offered to me.

Finally, I would like to thank my partner in life, Naomi, who had to put up with me throughout my studies at EPFL. Naomi, thank you very much for your patience, attention and love!

Contents

1 Introduction **1**

1.1 Introduction to BIP . . . 2

1.1.1 Behaviour . . . 2

1.1.2 Interaction . . . 3

1.1.3 Priority . . . 6

1.1.4 BIP implementations . . . 8

2 Overview **9**

2.1 System . . . 9

2.2 Behaviour layer . . . 9

2.2.1 Atoms . . . 10

2.2.2 Ports . . . 11

2.2.3 Awaiting on ports . . . 12

2.2.4 Spawning new atoms . . . 15

2.3 Glue layer . . . 16

2.3.1 Interactions . . . 16

2.3.2 Connectors . . . 16

3 Connector combinators **19**

3.1 Introduction to connectors . . . 19

3.2 Algebras of BIP . . . 20

3.2.1 Algebra of Interactions . . . 20

3.2.2 Algebra of Connectors . . . 21

3.2.3 Connector combinators . . . 22

3.3 Core combinators without data . . . 23

3.3.1 Grammar . . . 23

3.3.2 Semantics . . . 23

3.3.3 Relation to the algebras of BIP . . . 24

3.3.4 Algebraic properties . . . 24

3.3.5 Derived n-ary combinators . . . 24

3.4 Extensions to the theory of connectors . . . 25

3.4.1 Data transfer . . . 25

3.4.2 Types and semantic domains . . . 28

3.4.3 Grammar . . . 29

3.4.4 Denotational semantics 29
3.5 Connector combinators 32
3.5.1 Core combinators 32
3.5.2 Data manipulation combinators 34
3.5.3 Priority combinators 36
3.5.4 Dynamic combinators 38
3.5.5 Other derived combinators 39
3.6 Properties 41
3.6.1 Soundness 41
3.6.2 Algebraic properties 47
3.7 Properties from Category Theory 55
3.7.1 Hardness 62
3.7.2 Stability 66

4 Implementation **71**
4.1 General principles and architecture 72
4.1.1 Separation between user facing code and backend 72
4.1.2 Domain Specific Language 72
4.1.3 Engine architecture 74
4.1.4 Final words 77
4.2 Haskell implementation 78
4.2.1 Systems 78
4.2.2 Ports 81
4.2.3 Atoms 82
4.2.4 Connectors 84
4.2.5 Interactions 89
4.2.6 Waiting list 90
4.2.7 Semantic function 93
4.2.8 Engine 95
4.3 Scala implementation 104
4.3.1 Systems 104
4.3.2 Atom and ports 104
4.3.3 Atom actions 106
4.3.4 Connectors 109
4.3.5 Interactions 116
4.3.6 Waiting list 117
4.3.7 Semantic function 119
4.3.8 Engine 124

5 Optimisations **129**
5.1 Early detection of downwards incompatibility 130
5.2 Caching of interactions 136
5.2.1 Multiple layers of caching 138
5.2.2 Final words on caching 138

5.3 Connector optimisation . . . 139
5.3.1 Dead `Bind` elimination . . . 139
5.3.2 `Failure` and `Success` propagation . . . 140
5.3.3 Engine changes . . . 141
5.4 Early execution of stable interactions . . . 142

6 Examples and evaluation 151
6.1 Token ring (Haskell) . . . 151
6.1.1 Implementation . . . 151
6.1.2 Expressivity . . . 153
6.1.3 Performance . . . 153
6.2 Masters-Slaves (Scala) . . . 155
6.2.1 Implementation . . . 155
6.2.2 Expressivity . . . 157
6.3 Producers-Consumers (Haskell) . . . 158
6.3.1 Implementation . . . 158
6.3.2 Expressivity . . . 158
6.3.3 Performance . . . 159
6.4 Dining Philosophers (Haskell) . . . 161
6.4.1 Implementation . . . 161
6.4.2 Expressivity . . . 166
6.4.3 Performance . . . 166
6.5 Dining Philosophers (Scala) . . . 170
6.5.1 Implementation . . . 170
6.5.2 Expressivity . . . 173
6.5.3 Final words . . . 173

7 Discussion and Conclusion 175
7.1 Discussion . . . 175
7.1.1 Critics of the connectors . . . 175
7.1.2 Critics of atoms . . . 176
7.1.3 General observations . . . 176
7.1.4 Specificities of Haskell and Scala . . . 177
7.2 Future work . . . 178
7.2.1 Caching . . . 178
7.2.2 Analysis . . . 179
7.2.3 Distribution . . . 179
7.3 Conclusion . . . 179

Appendices 185

A Introduction to Category Theory 187
A.1 Category . . . 187
A.2 Functor . . . 187

A.3 Natural transformation 188
A.4 Monad 188

B Introduction to Haskell **191**
B.1 Basic syntax 191
B.2 Evaluation strategy 193
B.3 Purity 193
B.4 Declarations 193
B.5 Typing 194
B.6 Algebraic data types 195
B.7 Type classes 197
B.8 Computations with effects 199
B.8.1 The `IO` type constructor 199
B.8.2 The `main` action 200
B.8.3 The `Monad` instance of `IO` 200
B.9 Concurrency support 200
B.9.1 `forkIO` 200
B.9.2 `MVars` 201

Chapter 1

Introduction

From embedded systems to large distributed systems, concurrency is everywhere. Even in the world of general purpose systems, concurrency is becoming increasingly important. We can no longer rely on Moore's law[1] and wait for faster processors to come out at regular intervals. Instead of speeding up single processors, hardware manufacturers are shifting to the world of many-cores. With the multiplication of cores and processors in today's systems, concurrency is becoming a must to achieve reasonable performance. Parallelisation of sequential code performed by hardware is reaching its limits. Programs must be written with concurrency in mind to take advantage of those increasingly common multi-cores architectures[2].

Alas, writing concurrent systems is difficult. Going from sequential programs to systems of multiple concurrent components is not an easy task. The usual solution of using threads and synchronisation primitives such as locks, semaphores[3] and monitors[4] is highly error prone and unsatisfactory, due to their largely non-deterministic behaviour[5]. We are in need better of abstractions.

In this thesis, we introduce a formalism to describe concurrent systems derived from the Behaviour-Interaction-Priority framework (BIP)[6][7]. We have built a theoretical framework inspired from BIP to describe coordination between components, as well as some standard extensions such as data transfer and dynamicity.

In addition to this formalism, we show two implementations of BIP as domain specific languages embedded in two different functional programming languages: Haskell[8] and Scala[9]. We hope to obtain expressive frameworks to easily express systems of concurrent components directly in Haskell and Scala. We think that this embedding of BIP in functional programming languages leads to great improvements in the usability.

Before we proceed any further into this thesis, will we briefly introduce the concepts of BIP. This short introduction will highlight the fundamental concepts behind the approach.

1.1 Introduction to BIP

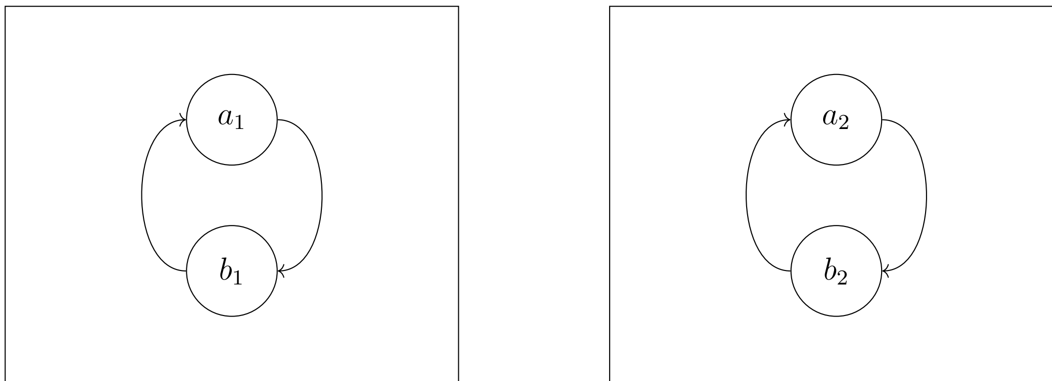
As we've just argued, writing concurrent programs is a difficult task. We believe most of the difficulty comes from the interleaving of computation and coordination usually present in most concurrent programs. The Behaviour-Interaction-Priority (BIP) approach tries to solve this problem by having a strict separation between computation and coordination code. We strongly believe this separation allows programmers and system designers to have a better understanding of both types of code (computation and coordination). This also make concurrent code more amenable to analysis[10].

In practice, BIP takes the form of an external domain specific language (DSL) and framework. It was originally developed in the context of embedded systems, and as since been adapted to work in different settings[11]. With this thesis, we bring and adapt the key concepts of BIP to functional programming languages.

As we have just said, the main idea of BIP is to have a strict separation between computation and coordination code. For this reason, BIP systems are decomposed into three distinct layers: Behaviour, Interaction and Priority. The computation code takes place in the Behaviour layer, and the coordination in the Interaction and Priority layers. The Interaction and Priority layers are sometimes called the *Glue* of the system.

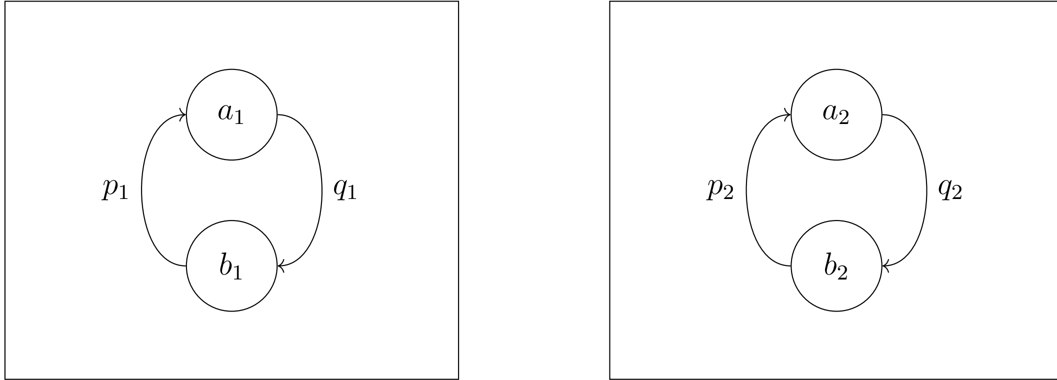
1.1.1 Behaviour

The first layer of BIP is the Behaviour layer. This layer is composed of a fixed number of heterogenous components, each defined in isolation. Those components are called *atoms* in the context of BIP. Each component can be modelled as a finite state machine (FSM). States in the state machine simply correspond to control locations, and transitions represent computations. As a first example, here are two atoms with the same behaviour.

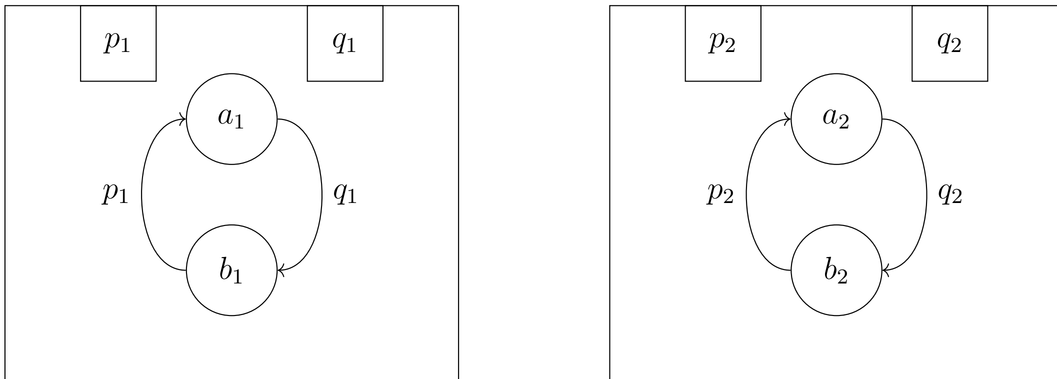


As we have previously stated, computations take place uniquely during transitions. States in the FSMs are simply control locations. In this model, when an atom reaches a control location, it stops its execution and waits for one of its possible transition to be triggered. It is important to note that transitions are not automatically triggered upon reaching the control location. An external entity, such as an engine, must notify the agent of which transition, if any, to take.

In order to refer to the transition of an atom, they are each given a label. We call these labels *ports*. Note that, in BIP, ports are unique to an atom. However, within the same atom, some transitions may share the same label given that they do not start in the same state.



Above is the previous example, with transitions labeled by a port. The ports of an atom form its interface to the rest of the system. This is generally represented in the following way:



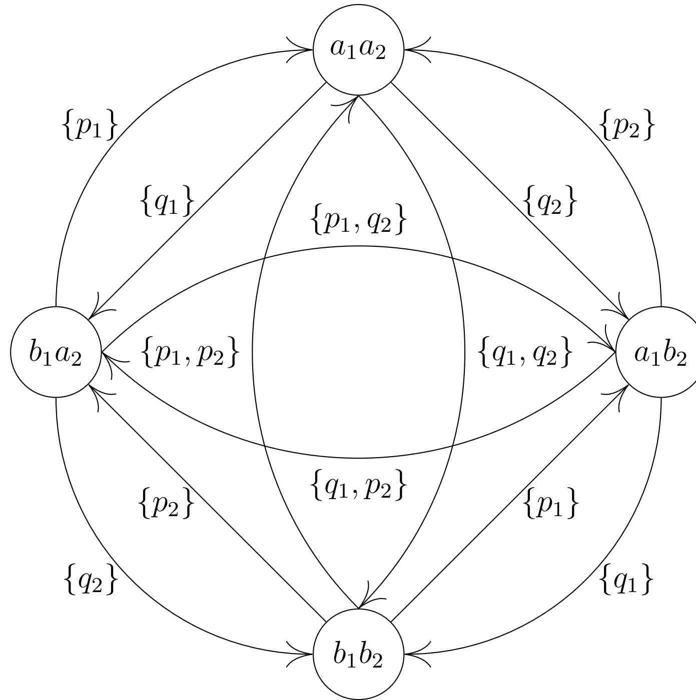
As can be seen from the above schema, the atoms of the system are at this point still not connected in any way. We will see in the next sections how those components can cooperate and coordinate.

1.1.2 Interaction

In the previous section, we looked at the Behaviour layer, which was focused on the behaviour of each individual atoms. We now move to the second layer of BIP, the Interaction layer. Contrary to the previous layer, the Interaction layer is concerned with the system as a whole.

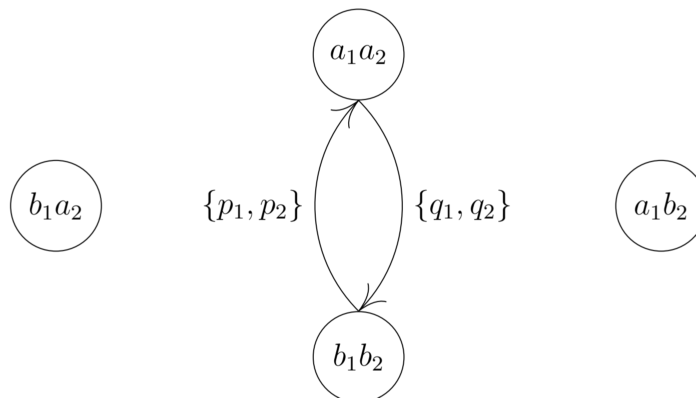
To begin our discussion of this layer, we need to look at the state machine of the entire system, which is simply the product of the state machines of every atoms. Specifically, the state machine of the system will have a state for every combination of states from its atoms. Transitions in the state machine correspond to subsets of

transitions of the underlying atoms. Below is shown the state machine of the system corresponding to the previous example.



Notice that the transitions in the state machine of the entire system are labeled with subset of ports. We will call such a subset of ports an *interaction*. The goal of the Interaction layer is to describe which of those interactions are possible. In the above state machine, the transitions have not been restricted in any way. Depending on the application however, this is not always desirable.

For instance, we might want the two atoms to always be in the same exact same states. In this case, we would like to only enable transitions that keep the two atoms in sync.

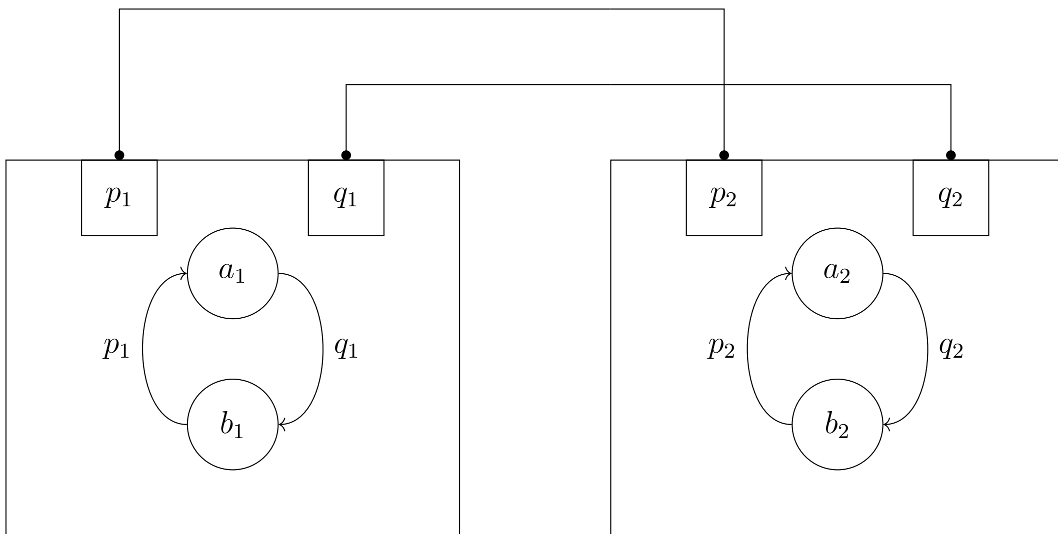


Connector

In the above example, the set of possible interactions was small enough so that they could be explicitly listed. However, in larger system, the set of possible interactions will become increasingly larger and explicitly listing them would be too long and error-prone.

The solution proposed by BIP to solve this problem is to use connectors[12]. A connector simply connects together the ports of the various atoms of a system, and in doing so, describes the possible interactions involving those ports. In a connector, each port is either given a *synchron* role, which specifies that the port may take part in the interaction, or a *trigger* role, which states that the port may initiate the interaction. For an interaction to be possible, either all connected ports must be enabled, or at least one of the triggers must be.

As a first example, here are the connectors which ensure that the two atoms from the previous example are always in the same states. The first connector states that p_1 and p_2 can synchronise, and the second that q_1 and q_2 can also synchronise. In the schema, triggers are represented with a triangle and synchrons with a circle.



The above schema defines two connectors. The first one connects the ports p_1 and p_2 and the second q_1 and q_2. Since all ports are given a synchron role, the only possible interactions are:

$$\{\{p_1, p_2\}, \{q_1, q_2\}\}$$

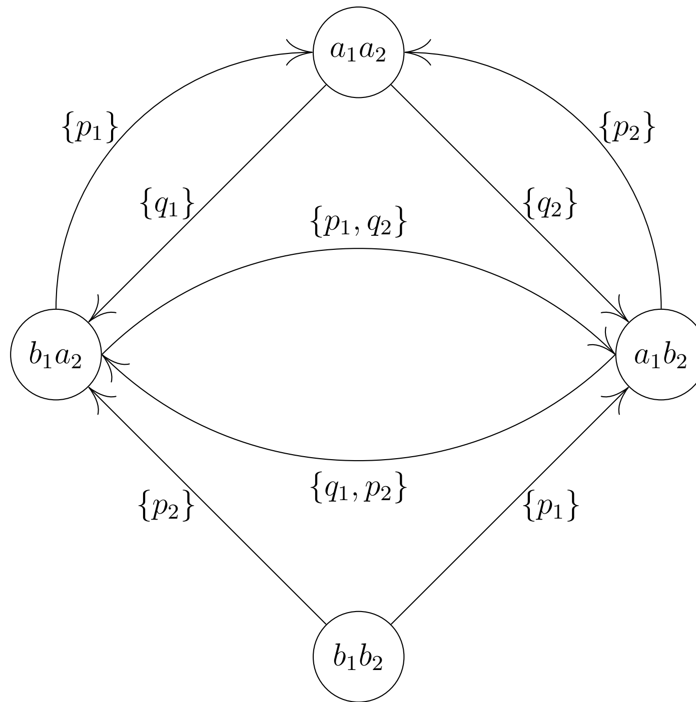
Which is exactly the subset of interactions we wanted to enable.

Note that connectors can also be hierarchically composed. Instead of connecting ports, connectors can also connect to other connectors, thereby creating a hierarchy of connectors. However, in the scope of this short introduction to BIP, we will not discuss this point in any more details.

1.1.3 Priority

As you may have noticed, some transitions in the state machine of a system will share the same label. Two different transitions in the state machine thus may correspond to the same interaction. Depending on the application, it might be desirable to enable one of the transitions and not the other. Since they are labeled with the same interaction, this is not feasible in the Interaction layer. The Priority layer of BIP is there for this purpose.

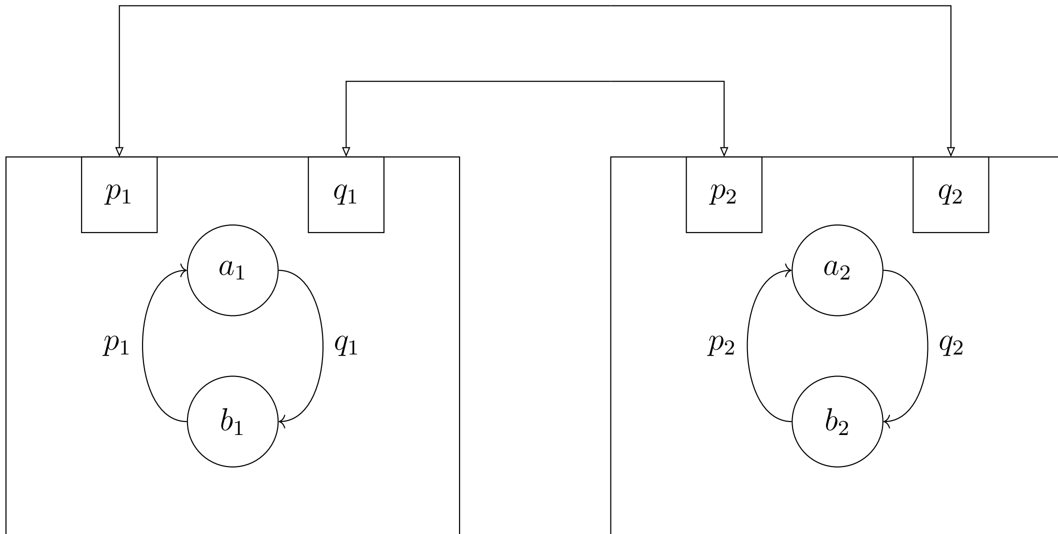
To illustrate this point, let us go back to our previous example and try to enforce the fact that the two atoms should never be in the state b at the same time. Below is shown the state machine of the complete system with all the transitions we wish to enable.



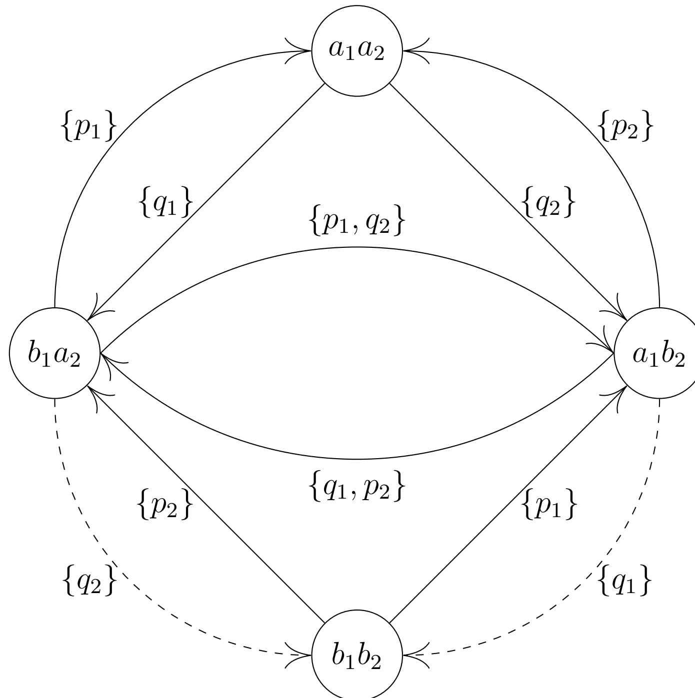
It is clear from the above schema that if we do not start in a forbidden state, then it is impossible to reach such a state later. As can be seen in the above schema, the following interactions should be possible:

$$\{\{p_1\}, \{p_2\}, \{q_1\}, \{q_2\}, \{p_1, q_2\}, \{q_1, p_2\}\}$$

This subset of interactions can be easily encoded using connectors, as shows the following schema.



The above connectors connect the port p_1 with q_2 and the port p_2 with q_1. Since they are all triggers, the connectors indeed describe the precise subset of interactions we wished to enable. However, if we look at the state machine of the system with the specified interaction model, some transitions that we wish to eliminate are still possible. Those transitions have been represented as dashed lines in the schema below.



Again note, and this is very important, that those transitions could not have been removed in the Interaction layer, since they share the same label with transitions that we wish to have.

In order to remove those interactions, a priority policy is to be applied. In the context of BIP, a priority policy is simply a partial order between the interactions.

In a given state, only the interactions with maximal priority amongst those currently possible according to the interaction model may be taken.

In our previous example, we can use the following priority policy to ensure that the two transitions we wish to disable are never taken:

$$\{q_1\} < \{q_1, p_2\}$$
$$\{q_2\} < \{p_1, q_2\}$$

Using this priority policy, the two transitions that were problematic and that could lead to the forbidden state have been removed.

1.1.4 BIP implementations

The theoretical framework that we have presented has been implemented in various ways, most notably as a complete tool chain which generates C++ code from the description of BIP systems in an external domain specific language[7][13]. In this thesis, we will present implementations of BIP as an embedded domain specific language in functional programming languages. In the next chapter, we will present an overview of the frameworks we have developed.

Chapter 2

Overview

In this chapter, we present a complete overview of the concepts and syntax of the Haskell and Scala frameworks. As you will see, the frameworks are conceptually very similar to BIP. We however differ in some crucial ways that we will explore. We believe however that the spirit of BIP is preserved!

Throughout this chapter, we illustrate the different concepts with actual code written in both the Haskell and Scala frameworks. Please note that the code samples presented here are there for illustration purposes mainly. For convenience, a very short introduction to the key concepts of Haskell can be found in the appendices.

2.1 System

The top level concept behind the frameworks in the concept of systems. As we will see, a system is composed of some concurrently executing components and some coordination strategy. Throughout this chapter, we will see how to describe such systems and execute them.

Just as in BIP, systems described with the frameworks are decomposed into distinct layers. Where BIP has three different layers, we, however, have only two. In our approach, we merge together the two glue layers, Interaction and Priority, into a single one. We will start our description of the frameworks with the first layer: the Behaviour layer.

2.2 Behaviour layer

As they are explicitly targeted at concurrency, the frameworks have naturally been designed to build concurrent systems. In such system, we make the distinction between the computation code, which generally performs the business logic, and the coordination code, which coordinates the different components. We argue that the two types of code are orthogonal. In the frameworks we designed, as in BIP, the two different kind of codes are kept as separate as possible.

The computation code is written in the Behaviour layer of the frameworks. The task to be performed is generally split between various components, which we call *atoms*.

2.2.1 Atoms

Atoms are active objects that can be assimilated to threads or actors[14] in other models, in the sense that they conceptually are concurrently executing processes.

In the frameworks, atoms are always bound to a specific system. Before a system actually starts executing, an initial set of its atoms must be declared. This is done in the following way.

```haskell
system = do
    atom <- newAtom someAction
    return ()
```

Haskell

```scala
val system = new System

val atom = system.newAtom(someAction)
```

Scala

The above programs simply create a system containing a single atom. The atom is responsible for the execution of the computation named `someAction`. `someAction` is not forced to terminate its execution, but if it does, the atom is said to die and is removed from the system.

Also note that the previous code snippets do not actually start executing the systems. To do so, the systems must be explicitly started.

```haskell
runSystem system
```

Haskell

```scala
system.run()
```

Scala

Given a system as we have just defined, the above instruction in Haskell and Scala will start executing the system. The instruction will block until the system has finished its whole execution, which, as we will see, arises either when all atoms have finished their execution or when the system enters a deadlock state, in which no interactions can be executed.

Hello World

At this point, we are in a position where we can actually show the first complete application. As is tradition, we present the code to run a system that displays the text `"Hello World!"`.

```haskell
runSystem $ do
    newAtom $ do
        liftIO $ putStrLn "Hello World!"
    return ()
```

Haskell

```scala
val system = new System
system.newAtom {
  println("Hello World!")
}
system.run()
```

Scala

The above programs, even though traditional, do not illustrate very well the capabilities of the frameworks. As the goal of the frameworks is to tackle concurrency, let us build a system with two atoms, each responsible to display a part of the message.

```
runSystem $ do
    newAtom $ do
        liftIO $ putStrLn "Hello!"
    newAtom $ do
        liftIO $ putStrLn "World!"
    return ()
```

Haskell

```
val system = new System
system.newAtom {
  println("Hello!")
}
system.newAtom {
  println("World!")
}
system.run()
```

Scala

The above programs create a system of two atoms, each of which is responsible to display a single message. When running the above programs, we run into the problem that, as the two atoms run concurrently, the messages may appear in the wrong order or even interleaved! This behaviour is something to be expected by design. Indeed, the whole point of a concurrency framework is to allow components to run concurrently, and find a way to coordinate them. We will shortly see how to perform the latter in the Haskell and Scala frameworks, when we present the Glue layer. To do so, we need first to introduce the concept of a *port*.

2.2.2 Ports

In the frameworks, ports are the way atoms can communicate and synchronise with other atoms. A port can be thought of as one the ends of some potentially complex communication channel which will be described in the Glue layer. This allows, as we will see, atoms to perform a single operation: sending a value into the channel and then receiving a value back in return. They are declared as part of a system as follows:

```
system = $ do
    port <- newPort
    -- Rest of the system.
    return ()
```

Haskell

```
val system = new System

val port = system.newPort
```

Scala

Ports have their type parameterised with the type of value they can send and the type of value they will receive. Those types can either be inferred from the context or explicitly specified, as the following example shows:

```haskell
system = $ do

    -- The first type, Int,
    -- indicates the type of values
    -- that the port can receive
    -- from the system.
    -- The second type, String,
    -- indicates the type of values
    -- that can be sent through the port.
    port :: PortId Int String <- newPort
    -- Rest of the system.
    return ()
```

Haskell

```scala
val s = new System

// The first type, Int,
// indicates the type of values
// that the port can receive
// from the system.
// The second type, String,
// indicates the type of values
// that can be sent
// through the port.
val port = s.newPort[Int, String]
```

Scala

Note that ports are not bound to a specific atom as they were in BIP. All ports can be used by all atoms of the system. They are more closely related to the BIP concept of port types. We will now see how the atoms make use of the ports.

2.2.3 Awaiting on ports

To use ports, atoms can use the `await` instruction, which sends a value into a port and wait for a value to be received. Here is first an illustration is Haskell:

```haskell
system = $ do
    -- Creating the port
    port <- newPort

    -- Creating the atom
    atom <- newAtom $ do
        -- Using the await instruction
        valueReceived <- await port valueToSend
        -- Some computation using the valueReceived
        someAction valueReceived

    -- Rest of the system.
    return ()
```

Similarly, the same example system could be written in Scala as follows:

```scala
val system = new System

// Creating the port
val port = system.newPort

// Creating the atom
```

```
val atom = system.newAtom {
  // Using the await instruction
  await(port, valueToSend) {
    case valueReceived => {
      // Some computation using the valueReceived
      someAction(valueReceived)
    }
  }
}
```

The `await` instruction is a conceptually blocking instruction. The atom must indeed wait to receive a value back from the port before its execution can continue. However, the implementations make sure that atoms waiting on ports do not block underlying threads or processes. For this reason, the continuation[1] of an atom must be obtainable, which is syntactically apparent in the Scala version, since the continuation is explicitly passed as an argument to the await function.

The previous instruction only allowed atoms to wait on a single port. It can be however desirable to send values to multiple ports and only wait for one of them to receive a value. This possibility is offered by the `awaitAny` instruction, which generalises `await`.

```
system = $ do
    -- Creating the two ports
    port1 <- newPort
    port2 <- newPort

    -- Creating the atom
    atom <- newAtom $ do
        -- Using the await instruction
        valueReceived <- awaitAny
            [ onPort port1 valueToSend1
            , onPort port2 valueToSend2 ]

        -- Some computation using the valueReceived
        someAction valueReceived

    -- Rest of the system.
    return ()
```

Which translates in Scala as:

```
val system = new System

// Creating the two ports
```

[1] A continuation is a function that returns the rest of some execution.

```
val port1 = system.newPort
val port2 = system.newPort

// Creating the atom
val atom = system.newAtom {
  // Using the awaitAny instruction
  awaitAny(port1 withValue valueToSend1, port2 withValue valueToSend2) {
    case valueReceived => {
      // Some computation using the valueReceived
      someAction(valueReceived)
    }
  }
}
```

Since users might want to distinguish the different cases, we offer a way to apply a different function on the potential value received by each port. This enables users to wrap the value received by different ports in different constructors, and latter pattern match on the wrapped value. For instance:

```
system = $ do
    -- Creating the two ports
    port1 <- newPort
    port2 <- newPort

    -- Creating the atom
    atom <- newAtom $ do
        -- Using the await instruction
        value <- awaitAny
            [ fmap Left  $ onPort port1 "Some string"
            , fmap Right $ onPort port2 0 ]

        case value of
            Left  valueReceived -> someAction valueReceived
            Right valueReceived -> someOtherAction valueReceived

    -- Rest of the system.
    return ()
```

Which translates in the Scala framework to:

```
val system = new System

// Creating the two ports
// Note that the type of values sent and received by the ports
// are explicitly specified.
```

```
val port1 = system.newPort[Int, String]
val port2 = system.newPort[Double, Long]

// Unfortunately the following functions are necessary here
// since type inference has its limits in Scala.
def left(x: Int): Either[Int, Double] = Left(x)
def right(y: Double): Either[Int, Double] = Right(x)

// Creating the atom
val atom = system.newAtom {
  // Using the awaitAny instruction
  awaitAny(port1 withValue "Some string" map left,
           port2 withValue 0 map right) {
    case Left(valueReceived)  => someAction(valueReceived)
    case Right(valueReceived) => someOtherAction(valueReceived)
  }
}
```

It is important to note that atoms have no direct control over the communication channel behind the ports. The communication channels, which connect the different ports of a system and are therefore called *connectors*, will be described in the Glue layer.

2.2.4 Spawning new atoms

The last point we wish to explore before we move to the exposition of the coordination of atoms is the notion of dynamicity. Traditionally in BIP, the set of atoms of a system is fixed before the system actually starts. It is thus impossible for new atoms to be created and later join the system. In the developed frameworks, we alleviate this restriction and allow atoms to spawn other atoms. This is performed using the `spawn` or `spawnAtom` instruction, as shown in the following example in Haskell.

```
system = $ do
    -- Rest of the system.
    newAtom $ do
        liftIO $ putStrLn "In the first atom."

        spawn $ do
            liftIO $ putStrLn "In the second atom."

        liftIO $ putStrLn "Still in the first atom."
```

Which can be written in Scala as:

```
val system = new System

system.newAtom {

  println("In the first atom.")

  spawnAtom {
    println("In the second atom.")
  }

  println("Still in the first atom.")
}
```

Note that we make a distinction between the creation of atoms before the system is started, which is performed using `newAtom`, and the spawning of atoms by other atoms while the system is running, which is done via `spawnAtom`. Indeed, the atoms created before the system is run can be explicitly referred to in the Glue layer. As we will see, the Glue layer is fixed before the system start executing. Therefore, at the time is it defined, the Glue layer has no knowledge of the atoms that will be spawned during the runtime of the system.

2.3 Glue layer

Above the Behaviour layer, which we have just described, comes the second and final layer. This second layer, called the Glue layer, precisely describes how the different atoms are coordinated. This last layer corresponds to a combination of the Interaction and Priority layers of BIP.

2.3.1 Interactions

The most important concept of the Glue layer is the concept of *interactions*. An interaction consists of a set of waiting atoms which synchronise and exchange values, each through some potentially different port. Before it can be executed, all atoms part of the interaction must be waiting on the port specified by the interaction. When the interaction is executed, all atoms that are part of the interaction receive a value on the selected port and can resume their execution.

2.3.2 Connectors

As systems may have many possible interactions, it would be very verbose and error-prone to explicitly list them all. In the frameworks, as in BIP, connectors are the way to concisely and precisely describe all possible interactions of a system. Connectors can be thought of as a way to connect together the different ports used by the different atoms of a system.

To describe connectors, we take a different approach than the one that is proposed in BIP. As we will see in details in the next chapter, connectors in the framework are built from a set of primitive combinators. Those combinators provide users with an expressive language in which connectors can be described.

The next chapter will be entirely dedicated to the combinators we use to describe the connectors. As a foretaste however, here is an example of a system with an associated connector. The system is composed of n atoms, which can either be working or idle. The connector of the system ensures that all atoms start and stop working at the same time.

```
system = $ do
    -- Creating the two ports
    start <- newPort
    end   <- newPort

    -- Number of atoms
    let n = 10

    -- Creating the atoms
    atoms <- replicateM n $ newAtom $ forever $ do
        await start ()
        putStrLn "Working!"
        await end ()
        putStrLn "Done!"

    -- Register the connector that
    -- specifies that all atoms must start and
    -- stop working at the same time
    registerConnector $ anyOf
        [ allOf [ bind atom start | atom <- atoms ]
        , allOf [ bind atom end   | atom <- atoms ] ]
```

Which translates in the Scala framework as follows:

```
val system = new System

// Creating the two ports
val start = system.newPort[Any, Unit]
val end   = system.newPort[Any, Unit]

// Number of atoms
val n = 10

// Creating the atoms
val atoms = for ( i <- 1 to n) yield system.newAtom {
```

```
  def loop() {
    await(start) { (_: Any) =>
      println("Working!")
      await(end) { (_: Any) =>
        println("Done!")
        loop()
      }
    }
  }

  loop()
}

// Register the connector that
// specifies that all atoms must start and
// stop working at the same time
system.registerConnector(anyOf(
  allOf(atoms map { _ bind start } : _*),
  allOf(atoms map { _ bind end } : _*)))
```

The way we build connectors is thoroughly formalised in the next chapter, in which we will discuss connector combinators and their properties.

Chapter 3

Connector combinators

3.1 Introduction to connectors

As we have already seen in the previous chapters, *BIP* systems are composed of multiple isolated components. The only way the various components, or atoms, can interact with each other is through their ports and a connector. The ports of an atom define its interface to the rest of the system. They specify which kind of data can be sent and received through them.

When an atom wants to exchange values and synchronise with other components, it must activate some of its ports, sending values through them. Then, the atom stops its execution and waits to be part of an interaction. When an interaction takes place, all components that are part of the interaction synchronise and receive some values on one of their activated ports. They are then free to continue their execution.

Connectors are used to describe the possible interactions of a system. Each system has some connectors which precisely describe how the various ports are connected. Each connector specifies which subsets of ports can synchronise together and how the values are exchanged.

In this chapter, we will investigate the theory behind connectors. We will begin our investigation with two algebras that are used to describe subsets of ports. We will then add data, priorities and dynamicity to our theoretical framework of connectors. This theoretical framework will serve as a basis to the functional implementations of BIP developed as part of this thesis.

3.2 Algebras of BIP

In *BIP*, components synchronise and communicate with each other through their ports. Once in a stable state, a system can execute an *interaction* to continue its execution. Ignoring the data transfer aspect, an interaction is simply a subset of ports which synchronise.

The possible interactions of a system, being simply sets of ports, can potentially be explicitly listed, but this representation is very verbose and error prone. In order to be able to concisely and conveniently describe the ways components can interact with each other, several algebras have been proposed. We will shortly introduce two of those algebras, namely the *Algebra of Interactions*[12] and the *Algebra of Connectors*[12].

3.2.1 Algebra of Interactions

The *Algebra of Interactions*, denoted by $\mathcal{AI}(P)$, gives us a concise way to describe subset of ports from P. Its grammar is given by:

```
<AI> ::= <p> | 1 | 0 | <AI> +  <AI> | <AI> * <AI>
```

for $\mathtt{p} \in P$.

Note that $*$, called fusion, has stronger precedence than $+$, called union. Also note that we will use parenthesis when necessary but for simplicity won't include them in the grammar. We will stick to this convention for grammars introduced later in this chapter as well.

The semantics of $\mathcal{AI}(P)$, which turns formulas into sets of ports, is given by:

$$
\begin{aligned}
[\![p]\!] &= \{\{p\}\} \\
[\![1]\!] &= \{\emptyset\} \\
[\![0]\!] &= \emptyset \\
[\![c_1 + c_2]\!] &= [\![c_1]\!] \cup [\![c_2]\!] \\
[\![c_1 * c_2]\!] &= \{i_1 \cup i_2 \mid i_1 \in [\![c_1]\!] \wedge i_2 \in [\![c_2]\!]\}
\end{aligned}
$$

Each $\gamma \in [\![c]\!]$ corresponds to an interaction. We say that a port $p \in P$ is part of an interaction γ if and only if $p \in \gamma$.

Examples Let the set of ports P be equal to $\{p, q\}$. The following holds:

$$
\begin{aligned}
[\![p]\!] &= \{\{p\}\} \\
[\![p + q]\!] &= \{\{p\}, \{q\}\} \\
[\![p + 1]\!] &= \{\{p\}, \emptyset\} \\
[\![p + 0]\!] &= \{\{p\}\} \\
[\![p * q]\!] &= \{\{p, q\}\} \\
[\![p * 1]\!] &= \{\{p\}\} \\
[\![p * 0]\!] &= \emptyset
\end{aligned}
$$

Algebraic properties We will consider $c_1, c_2 \in \mathcal{AI}(P)$ to be equal if and only if they correspond to the same set of interactions according to the semantics.

$$c_1 = c_2 \iff [\![c_1]\!] = [\![c_2]\!] \qquad \forall c_1, c_2 \in \mathcal{AI}(P)$$

Equipped with the above equality, $\mathcal{AI}(P)$ forms a *idempotent commutative semiring*. Precisely, this means that, for any $c_1, c_2, c_3 \in \mathcal{AI}(P)$, the following hold:

1. $(\mathcal{AI}(P), +, 0)$ is a commutative monoid:
   - $c_1 + (c_2 + c_3) = (c_1 + c_2) + c_3$
   - $c_1 + c_2 = c_2 + c_1$
   - $c_1 + 0 = c_1$
2. $(\mathcal{AI}(P), *, 1)$ is a commutative monoid:
   - $c_1 * (c_2 * c_3) = (c_1 * c_2) * c_3$
   - $c_1 * c_2 = c_2 * c_1$
   - $c_1 * 1 = c_1$
3. Multiplication, or fusion, distributes over addition:
   - $c_1 * (c_2 + c_3) = (c_1 * c_2) + (c_1 * c_3)$
4. Multiplication by 0 annihilates $\mathcal{AI}(P)$:
   - $c_1 * 0 = 0$
5. Addition, or union, is idempotent:
   - $c_1 + c_1 = c_1$

3.2.2 Algebra of Connectors

In the *BIP* framework, a different algebra is used to define the possible interactions of connectors, namely the *Algebra of Connectors*[12]. In addition to union and fusion, the algebra revolves around the concept of *triggers* and *synchrons*. Its grammar is given by:

```
<t>  ::= [0]  | [1]  | [<p>]  | [<AC>]     // Triggers
<s>  ::= [0]' | [1]' | [<p>]' | [<AC>]'    // Synchrons
<AC> ::= <t> | <s> | <AC> + <AC> | <AC> * <AC>
```

for $\mathtt{p} \in P$.

The Algebra of Connectors is usually given its semantics in terms of the Algebra of Interactions we have just seen. The following specifies that ports and union in the Algebra of Connectors corresponds to the equivalent concepts in the Algebra of Interactions.

$$\begin{aligned} [\![[p]]\!] &= p \\ [\![[p]']\!] &= p \\ [\![x + y]\!] &= [\![x]\!] + [\![y]\!] \end{aligned}$$

The interesting part is in the treatment of multiplication, or fusion, which differs from its treatment in the Algebra of Interactions.

$$[\![\prod_{i=1}^{n}[x_i]]\!] = \prod_{i=1}^{n}[\![x_i]\!]$$

$$[\![\prod_{i=1}^{n}[x_i]' * \prod_{j=1}^{m}[y_j]]\!] = \sum_{i=1}^{n}([\![x_i]\!] * \prod_{k \neq i}(1 + [\![x_k]\!]) * \prod_{j}(1 + [\![y_j]\!]))$$

Intuitively, it means that for the fusion of many connectors to be enabled, either:

- all connectors are enabled, or
- at least one of the triggers is enabled.

This algebra is very interesting since it allows programmers and designers to intuitively and concisely describe sets of interactions. We will aspire to this kind of expressivity for the connectors in our frameworks. This algebra is however of little interest for the rest of this chapter, since we will mostly base our formalisation on the Algebra of Interactions.

3.2.3 Connector combinators

To describe connectors in this thesis, we will not use any of the above algebras but a different approach, namely *connector combinators*. In addition to specifying the possible interactions in terms of involved ports, the connector combinators will allow us to precisely describe data transfer and priorities directly in the connectors. In addition, this approach has the advantage to be very easily mapped to functional programming constructs, namely algebraic data types, as we will see in the later chapters.

Throughout this chapter, we will introduce all connector combinators progressively, by groups of related combinators. Semantics, properties, typing rules as well as relations to other algebras of *BIP* will be provided.

Before diving into the complete set of connector combinators, let's introduce a restricted subset of it, called *core combinators without data*. This restricted subset is very closely related to the other algebras of *BIP*.

3.3 Core combinators without data

The first set of combinators we introduce are called *core combinators*. For simplicity, we will ignore data transfer for now and first introduce a variant of the core combinators without data. We will denote this set by $C_{\text{core}}(P)$, where P is the set of ports. As we will see, the core combinators correspond exactly to the *Algebra of Interactions*.

As before, a semantic function gives connectors a meaning in term of set of interactions, each interaction being for now simply a set of ports which synchronise. The core combinators are composed of the following primitive connectors:

- Ports p, which denotes the connection to a given port.
- `Success`, which corresponds to an always succeeding connector that always has an interaction. However, this interaction does not involve any port.
- `Failure`, which corresponds to an always failing connector, which never has any possible interaction.

In addition to those primitive connectors, the following combinators are defined:

- `OneOf` which expresses non-deterministic choice between two connectors.
- `BothOf` which expresses synchronisation between two connectors.

3.3.1 Grammar

The grammar of $C_{\text{core}}(P)$ is given by:

```
<CoreC> ::= <p>
          | Success
          | Failure
          | OneOf <CoreC> <CoreC>
          | BothOf <CoreC> <CoreC>
```

where `p` $\in P$.

3.3.2 Semantics

The goal of connectors is to describe the possible interactions of a system. The exact set of interactions described by a connector is given by the $[\![\cdot]\!] : C_{\text{core}}(P) \to 2^{2^P}$ function.

$$
\begin{aligned}
[\![p]\!] &= \{\{p\}\} \\
[\![\texttt{Success}]\!] &= \{\emptyset\} \\
[\![\texttt{Failure}]\!] &= \emptyset \\
[\![\texttt{OneOf}\ c_1\ c_2]\!] &= [\![c_1]\!] \cup [\![c_2]\!] \\
[\![\texttt{BothOf}\ c_1\ c_2]\!] &= \{i_1 \cup i_2 \mid i_1 \in [\![c_1]\!] \wedge i_2 \in [\![c_2]\!]\}
\end{aligned}
$$

for $p \in P, c_1, c_2 \in C_{\text{core}}(P)$.

3.3.3 Relation to the algebras of BIP

In their data-less version, the core combinators correspond exactly to the *Algebra of Interactions* and the *Algebra of Connectors*. In fact, there is a trivial isomorphism between $C_{\text{core}}(P)$ and $\mathcal{AI}(P)$.

	$C_{\text{core}}(P)$	$\mathcal{AI}(P)$
Ports	p	p
Additive neutral	`Failure`	0
Multiplicative neutral	`Success`	1
Addition	`OneOf` c_1 c_2	$x_1 + x_2$
Multiplication	`BothOf` c_1 c_2	$x_1 * x_2$

3.3.4 Algebraic properties

As the core combinators are isomorphic to the *Algebra of Interactions*, the properties of the *Algebra of Interactions* apply directly. This means that core combinators also form an idempotent and commutative semiring.

3.3.5 Derived n-ary combinators

The two binary combinators `BothOf` and `OneOf` can be generalised to n-ary combinators that we will call `AllOf` and `AnyOf`. They are defined recursively as:

$$\begin{aligned}
\texttt{AllOf}\ <> &= \texttt{Success} \\
\texttt{AllOf}\ <x_1, x_2, \ldots, x_n> &= \texttt{BothOf}\ x_1\ (\texttt{AllOf}\ <x_2, \ldots, x_n>)
\end{aligned}$$

$$\begin{aligned}
\texttt{AnyOf}\ <> &= \texttt{Failure} \\
\texttt{AnyOf}\ <x_1, x_2, \ldots, x_n> &= \texttt{OneOf}\ x_1\ (\texttt{AnyOf}\ <x_2, \ldots, x_n>)
\end{aligned}$$

The algebraic properties of the core combinators justify the definition of those two derived connectors. `AnyOf` is simply the n-ary sum operator and `AllOf` the n-ary product operator.

3.4 Extensions to the theory of connectors

So far, we have completely ignored the data transfer aspect of interactions. As you have seen, data transfers are not part of the *Algebra of Connectors* nor the *Algebra of Interactions*. In practice however, components of BIP systems must somehow exchange data. In the BIP framework, data transfers are usually performed using mutation of some internal state during interactions and are hidden in the algebras seen so far. In our theory of connectors, we will take a different approach and explicitly describe the data transfer directly within the connectors.

Priorities will also be added to our theory of connectors. As a reminder, priorities are used to filter out interactions based on the availability of other interactions. Priorities are not directly part of the previously seen theories and are typically added as an extra layer on top on the Interaction layer. In this thesis, we will directly describe priorities within the connectors.

Finally, we will also introduce combinators to take into account the dynamic nature of systems in which new atoms can be created. Even though the connector of the system is fixed, we would like it to be able to refer to atoms that have been spawned during the runtime of the system. Without those combinators, we would have no way to involve such atoms in any interaction. We call these combinators the *dynamic* combinators.

To support for these extensions, we will introduce:

1. A set of new connector combinators, called $\mathcal{C}$, whose grammar will be introduced progressively.

2. A very simple type system, to rule out connectors that might perform invalid operations on the data. We will only consider connectors to be valid, and thus part of $\mathcal{C}$, if they are typable.

3. A new denotational semantics function that also specifies how data is exchanged between the various ports of the various atoms and what are the priorities between the different interactions.

But first, let us develop an intuition on how data transfer actually happen within connectors.

3.4.1 Data transfer

In this section, we will discuss how data transfers are performed within the *BIP* framework. Then, we will present the different approach taken in this thesis.

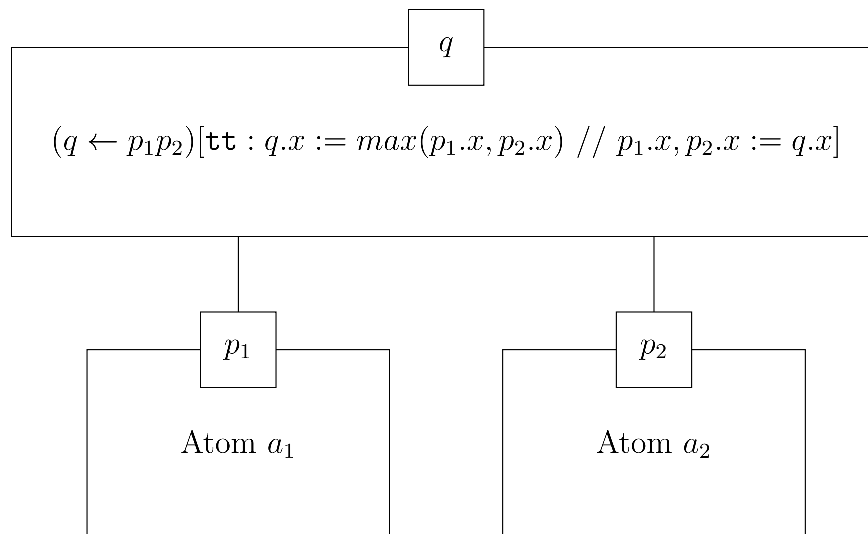
Data transfer in *BIP*

As we recall, in the *BIP* framework, the various components of a system, called atoms, synchronise and communicate via the help of a connector. The goal of the connector is to connect together in some specific way the ports of the various atoms. When an atom wants to synchronise and communicate, it stops its execution and waits on some

of its ports. We call a port on which an atom is waiting an *active* port. The values to be sent to the other components are then stored in the internal variables of the various active ports. The connector, given complete information about the state of each port, lists the possible interactions. The various interactions specify which atoms can continue their execution, if any, and possibly modifies the internal variables of the ports involved, thereby sending values back to the atoms.

Example Let's consider the following *BIP* system. The system consists of two atoms, each having a single port. All the ports of the system are connected together using a connector, which will describe the legal interactions and perform the data transfer. For this example, the only interaction possible is when both ports are active, in which case both ports should receive the maximum of the two values sent through the ports.

Note that we will use the notation introduced in [15] to describe the interaction described by the connector. For this very simple example, the notation should be very straightforward.



The connector specifies the possible interactions, in this case only one, and describes how data flows. As the task of describing interactions and associated data transfer policies can be very complex, the connectors in *BIP* are often hierarchically composed of multiple simpler connectors. For this reason, connectors, just as atoms, can export ports and thus be connected to other connectors.

Alternative approach

In this thesis, we take a different approach regarding connectors and data transfer. We make several major changes:

1. Ports no longer have internal variables. Instead, during an interaction, ports will be able to send a value to and receive a value from connectors just above them in the hierarchy of connectors.

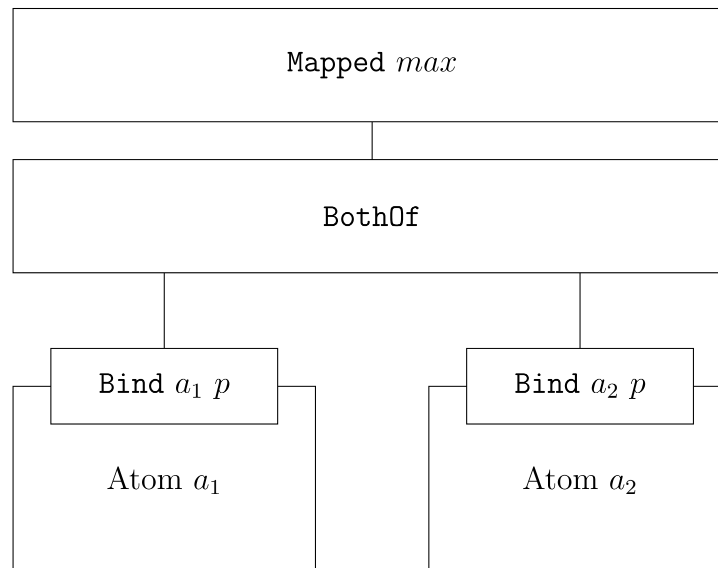
2. Connectors no longer have exported ports. Instead, connectors, just as ports, will be able during an interaction to send a value to and receive a value from the connector just above them. In addition, the connectors will also be able to send values to and receive values from connectors and ports just below them in the hierarchy of connectors.

3. Ports are no longer bound to a single atom. We consider ports to simply be entry points to connectors, which multiple atoms can use. Ports can be thought of as the port types of BIP, except that they have a single instance in each atom.

To make effective use of this simpler interface, we will need to use simpler basic connectors and rely heavily on hierarchical composition. As we will see, our primitive connector combinators will only achieve a single simple task. Complex connectors will be obtained through composition of simpler connectors.

As we will shortly see with an example, the values sent through the various ports will flow upwards through the hierarchy of connectors, during what is called the *upwards* phase. Then, when a value reaches the top level connector, it is sent back down the hierarchy of connectors until each involved port is reached. This phase is called the *downwards* phase.

Example As a foretaste, here is how the connector in the previous example could be defined:



As you can see, the connector is composed of simpler primitive connectors, all of which will be formally introduced later in this chapter. For the moment, we can get an intuition on how the various connectors work by walking through a step by step execution of an interaction for this particular example system:

1. The execution starts with an *upwards* phase, in which values flow up the hierarchy of connectors.

(a) The different active ports send a value upwards to the `BothOf` connector. Inactive ports do not send any values.

(b) The `BothOf` connector checks if it has received two values. If it is the case, then the connector sends upwards the two values as a pair to the `Mapped` connector. Otherwise, it does not send a value upwards.

(c) The `Mapped` connector, when it receives a value, simply applies a function to it and sends it upwards. The function applied in this particular example is the *max* function, which simply returns the maximal element of a pair. In case no value is received, this connector will not send anything upwards.

2. If a value reaches the top of the hierarchy of connector and is propagated upwards by the top level connector, in this case `Mapped`, then an interaction is possible. The value is then sent back to the connector and propagated downwards. This phase is called the *downwards* phase.

(a) When `Mapped` receives a value from upwards, it simply sends it to the connector directly below, `BothOf` in this case.

(b) The `BothOf` connector then receives a value, which it propagates downwards to both underlying ports.

(c) The involved ports then receive the value, and the respective atoms can continue their execution.

In the later sections we will see in detail how to define such connectors using *connector combinators*. But first, we will need to introduce several important concepts.

3.4.2 Types and semantic domains

In order to talk about data and valid manipulation of data, we will need a set of values and a type system in place. We will denote our set of values by $\mathcal{V}$. For each type a, we have a corresponding semantic domain, denoted by $sem(a)$, with $sem(a) \subseteq \mathcal{V}$. Conversely, each and every value $v \in \mathcal{V}$ will have at least a corresponding type a. We denote the fact that a value v has type a by $v : a$.

We consider the following types and semantic domains:

Name	Type	Semantic domain
Integers	`Int`	$\mathbb{Z}$
Booleans	`Bool`	$\{0, 1\}$
Functions	$a \to b$	$sem(a) \to sem(b)$
Tuples	$a \times b$	$sem(a) \times sem(b)$
Sequences of size n	$<a>_n$	$[1, n] \to sem(a)$

Ports As discussed before, values can be sent and received by ports. We restrict ports to send only a single type of values and receive a single, but possibly different, type of values. If p *sends value of type* u, we write $sendType(p) = u$ and, similarly, if p *accepts value of type* d, we write $receiveType(p) = d$.

Connectors In addition to the types presented above, we introduce an extra type for connectors and ports, denoted by $u \uparrow\downarrow d$, where u and d are types. A connector or port of type $u \uparrow\downarrow d$ will send upwards values of type u during the upwards phase and will accept values of type d during the downwards phase.

In this formalisation, we will consider connectors to be values in $\mathcal{V}$. Connectors can be used in the same context as any other values. For instance, connectors may be applied to functions or even be sent as upwards values given the appropriate types!

It also means that we will only consider connectors that are typable. All "connectors" described by the grammar that are not correctly typable won't be included in $\mathcal{C}$.

We will introduce the typing rules of connector combinators progressively, as soon as we encounter them.

Polymorphism It is possible that some values in $\mathcal{V}$ and some connectors in $\mathcal{C}$ will have multiple valid types. We say that these values and connectors are polymorphic.

3.4.3 Grammar

As explained before, the various combinators will be introduced in groups throughout the rest of this chapter. The *core combinators* will first be introduced, then the *data manipulation combinators* and the *priority combinators*. Finally, the *dynamic* combinators will be introduced. This is reflected in the grammar of the connectors:

```
<C> ::= <CoreC> | <DataC> | <PriorityC> | <DynamicC>
```

Where `CoreC`, `DataC`, `PriorityC` and `DynamicC` will be introduced later in this chapter.

3.4.4 Denotational semantics

The denotational semantics we have used so far for $\mathcal{AI}(P)$, $\mathcal{AC}(P)$ and $C_{\text{core}}(P)$ only showed the possible interactions in terms of involved ports and did not take into account data flow and data manipulation. We will shortly present a new denotational semantics function to precisely describe the data transfer that takes place during interactions, but first we will introduce some useful concepts and notation.

Partial functions We denote by $A \not\rightarrow B$ the set of partial functions from the set A to the set B. We will sometimes use the fact that partial functions can be represented as sets of pairs from $A \times B$, in particular when we will construct such partial functions. For instance, we will denote the empty partial function by $\emptyset$ and the singleton partial function that maps x to y by $\{(x, y)\}$ or even $\{x \mapsto y\}$.

Atoms We denote by $\mathcal{A}$ the set of atom identifiers. We consider that each atom of any system can be identified by a unique element of $\mathcal{A}$.

Ports We denote by P the set of ports of a system. Remember that ports are no longer bound to a single atom. We consider that each atom can possibly wait on every single port.

System states We will represent system states $\mathcal{S}$ as partial functions from atom-port pairs to the values possibly sent by the port, namely:

$$\mathcal{S} = \{f \in (\mathcal{A} \times P) \nrightarrow \mathcal{V} \mid \forall((a, p) \mapsto x) \in f \ . \ x \in \mathit{sem}(\mathit{sendType}(p))\}$$

The intuition behind this is that in a given global state, the various atoms may or may not wait on the different ports. When a port is active for an atom, a value of the appropriate type has been sent through it.

Assignment Similarly, we define *assignments* $\mathcal{R}$ to be a partial functions from atom-ports to the values that can possibly be received by the port. More precisely:

$$\mathcal{R} = \{f \in (\mathcal{A} \times P) \nrightarrow \mathcal{V} \mid \forall((a, p) \mapsto x) \in f \ . \ x \in \mathit{sem}(\mathit{receiveType}(p))\}$$

An assignment maps each atom-port pair to at most a single value. Intuitively, the value assigned, if any, is the value received by the atom on the given port.

Open interactions We denote by $\mathcal{O} = \mathcal{V} \times (\mathcal{V} \nrightarrow \mathcal{R})$ the set of open interactions. An open interaction consists of an *upwards value* in $\mathcal{V}$ and a function that when given a *downwards value* may return an assignment. We call this function the *downwards function* of an interaction. Those interactions are called *open* as the values exchanged are exposed.

We can close an open interaction to obtain an assignment using the *close* function:

$$\begin{aligned} &\mathit{close} : \mathcal{O} \to \mathcal{R} \\ &\mathit{close}((u, g)) = g(u) \end{aligned}$$

Closing an open interaction simply uses the upwards value as the downwards value.

Downwards compatibility We consider two assignments $R_1, R_2 \in \mathcal{R}$ to be *downwards compatible* if and only if:

$$\{a \mid (a, p) \in \mathit{domain}(R_1)\} \cap \{a \mid (a, p) \in \mathit{domain}(R_2)\} = \emptyset$$

Intuitively, two assignments are *downwards compatible* if they don't send values to the same atoms.

We can extend this notion to downwards functions as well. We consider two downwards functions g_1 and g_2 to be downwards compatible if and only if they don't send

values to the same atoms. More formally, this means that the downwards functions g_1, g_2 are *compatible* if and only if:

$$\forall x. \{a \mid (a, p) \in domain(g_1(x))\} \cap \{a \mid (a, p) \in domain(g_2(x))\} = \emptyset$$

We will later prove that, regardless of the downwards value, the domain of the resulting assignment is always the same for downwards functions obtained through the semantics function that we will define.

Finally, we can extend this notion to open interactions as well. We consider two open interactions to be downwards compatible if and only if the two downwards functions are downwards compatible.

Semantic functions We introduce a semantic function $[\cdot] : \mathcal{C} \to \mathcal{S} \to 2^{\mathcal{O}}$. This semantic function will be introduced progressively as we encounter new connector combinators. The semantic function will return, for each connector and system state, possibly multiple open interactions.

We also introduce the $[\![\cdot]\!] : \mathcal{C} \to \mathcal{S} \to 2^{\mathcal{R}}$ function, called the closed semantics. This semantics function is defined in terms of $[\cdot]$ as follows:

$$[\![c]\!](S) = \{close(o) \mid o \in [c](S)\}$$

3.5 Connector combinators

In this section, we introduce the combinators that will be used to construct connectors. As previously discussed, they will be introduced in groups of related combinators. We will first see the core combinators, then the data combinators, the priority combinators and finally the dynamicity combinators.

Remark. You will certainly notice that some of the combinators we introduce as primitives can be derived from other combinators, in a more or less straightforward way. There is a compromise to be found between keeping the number of combinators to a minimum for simplicity and having an intuitive set of comprehensible combinators. The combinators we have chosen as primitives represent, we think, a good compromise between those two goals.

3.5.1 Core combinators

The first set of combinators we introduce are the core combinators, which were previously introduced in their data-less version. With the introduction of data, we will consider once again those combinators.

Grammar

The core combinators are defined by the following grammar:

```
<CoreC> ::= Bind <a> <p>
          | Success <v>
          | Failure
          | OneOf <C> <C>
          | BothOf <C> <C>
```

for `a` $\in \mathcal{A}$, `p` $\in P$ and `v` $\in \mathcal{V}$.

Difference with data-less core combinators

The above grammar is very similar to the one for data-less core combinators. The differences are that:

- `Bind` is introduced in place of "naked" ports. Again, the reason is that now multiple atoms can connect to the same port. `Bind` allows us to talk about a port on a specific atom.
- `Success` is tagged with a value in $\mathcal{V}$. This value is the value that will be propagated upwards.
- Underlying connector combinators are not restricted to core combinators.

Intuition behind the combinators

The combinators can be understood in the following intuitive terms:

- `Bind` connects an atom to a port. The upwards value will be the value sent through the port by the atom.
- `Success` represents a always enabled connector which does not involve any port.
- `Failure` represents a connector that is never enabled.
- `OneOf` indicates non-deterministic choice between two connectors.
- `BothOf` indicates synchronisation between two connectors. The upwards value consists of the pair of underlying upwards values.

Typing rules

Below are given the typing rules for the core combinators:

$$\frac{}{\texttt{Bind}\ a\ p : \mathit{sendType}(p) \uparrow\downarrow \mathit{receiveType}(p)}$$

$$\frac{x : u}{\texttt{Success}\ x : u \uparrow\downarrow d} \qquad \frac{}{\texttt{Failure} : u \uparrow\downarrow d}$$

$$\frac{c_1 : u \uparrow\downarrow d \quad c_2 : v \uparrow\downarrow d}{\texttt{BothOf}\ c_1\ c_2 : (u \times v) \uparrow\downarrow d} \qquad \frac{c_1 : u \uparrow\downarrow d \quad c_2 : u \uparrow\downarrow d}{\texttt{OneOf}\ c_1\ c_2 : u \uparrow\downarrow d}$$

Observe that `Success` is polymorphic in the receive type and that `Failure` is polymorphic in both the send and receive type.

Semantics

The semantics of the core combinators is given below. Note that the semantics function is applied to both a connector and a system state $S \in \mathcal{S}$ in the definitions below.

$$
\begin{aligned}
[\texttt{Bind}\ a\ p](S) &= \begin{cases} \{(S(a,p), \{x \mapsto \{(a,p) \mapsto x\} \mid x \in \mathit{sem}(\mathit{receiveType}(p))\})\} \\ \qquad\qquad\quad \text{if } (a,p) \in \mathit{domain}(S) \\ \emptyset \qquad\qquad\quad \text{otherwise} \end{cases} \\
[\texttt{Success}\ v](S) &= \{(v, \lambda d.\emptyset)\} \\
[\texttt{Failure}](S) &= \emptyset \\
[\texttt{OneOf}\ c_1\ c_2](S) &= [c_1](S) \cup [c_2](S) \\
[\texttt{BothOf}\ c_1\ c_2](S) &= \{((x_1, x_2), \{x \mapsto g_1(x) \cup g_2(x) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\}) \\
&\quad \mid (x_1, g_1) \in [c_1](S) \\
&\quad \wedge (x_2, g_2) \in [c_2](S) \\
&\quad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\}
\end{aligned}
$$

Note on `OneOf` We can observe that `OneOf` introduces non-deterministic choice. Without `OneOf`, the semantics function could return at most a single open interaction. We will later see another connector combinator which also produces more than one interaction, namely `Dynamic`.

3.5.2 Data manipulation combinators

The next combinators we introduce allow us to manipulate the data flowing through the connectors. We introduce three new combinators:

- `Mapped`, which applies a function to the upwards value.
- `ContraMapped`, which applies a function to the downwards value.
- `Guarded`, which ensures that a predicate holds on the upwards value.
- `Feedback`, which feeds upwards values downwards.

Grammar

The grammar rules for the three data manipulation combinators are introduced below.

```
<DataC> ::= Mapped <f> <C>
          | ContraMapped <f> <C>
          | Guarded <f> <C>
          | Feedback <C>
```

for $f \in \mathcal{V}$.

Typing rules

Their typing rules are as follow:

$$\frac{f : u \to v \qquad c : u \uparrow\downarrow d}{\texttt{Mapped}\ f\ c : v \uparrow\downarrow d} \qquad \frac{f : e \to d \qquad c : u \uparrow\downarrow d}{\texttt{ContraMapped}\ f\ c : u \uparrow\downarrow e}$$

$$\frac{f : u \to \texttt{Bool} \qquad c : u \uparrow\downarrow d}{\texttt{Guarded}\ f\ c : u \uparrow\downarrow d} \qquad \frac{c : u \uparrow\downarrow (d, u)}{\texttt{Feedback}\ c : u \uparrow\downarrow d}$$

Semantics

The denotational semantics of the three combinators is given by:

$$\begin{aligned}
[\texttt{Mapped}\ f\ c](S) &= \{(f(x), g) \mid (x, g) \in [c](S)\} \\
[\texttt{ContraMapped}\ f\ c](S) &= \{(x, g \circ f) \mid (x, g) \in [c](S)\} \\
[\texttt{Guarded}\ f\ c](S) &= \{(x, g) \mid (x, g) \in [c](S) \wedge f(x) = 1\} \\
[\texttt{Feedback}\ c](S) &= \{(x, g \circ tag_x) \mid (x, g) \in [c](S)\}
\end{aligned}$$

Where the tag_x function is defined as:

$$tag_x(y) = (y, x)$$

We can clearly see that `Mapped` and `ContraMapped` simply apply functions to respectively upwards and downwards values. `Guarded` reduces non-determinism by filtering out interactions which do not satisfy a given predicate.

Remark. The names `Mapped` and `ContraMapped` have been chosen for those combinators as, as we will see, they correspond to similarly named concepts in functional programming languages and category theory.

Derivation of `AllOf` and `AnyOf`

Using the above core and data manipulation combinators, we can yet define some useful derived combinators. Let's introduce `AllOf` and `AnyOf`, which are the n-ary equivalent of respectively `BothOf` and `OneOf`. They can be recursively defined as:

$$\begin{aligned}
\texttt{AllOf}\ {<}{>} &:= \texttt{Success}\ {<}{>} \\
\texttt{AllOf}\ {<}c_1\ c_2\ \ldots\ c_n{>} &:= \texttt{Mapped}\ \mathit{cons}\ (\texttt{BothOf}\ c_1\ (\texttt{AllOf}\ {<}c_2\ \ldots\ c_n{>}))
\end{aligned}$$

Where $cons : (u \times \langle u \rangle_n) \to \langle u \rangle_{n+1}$ is the functions that appends an element in front of a sequence.

$$\texttt{AnyOf} <> := \texttt{Failure}$$
$$\texttt{AnyOf} <c_1\ c_2\ \dots\ c_n> := \texttt{OneOf}\ c_1\ (\texttt{AnyOf} <c_2\ \dots\ c_n>)$$

Their typing rules are:

$$\frac{c_i : u \uparrow\downarrow d \quad \forall i \in [1,n]}{\texttt{AllOf} <c_1\ c_2\ \dots\ c_n> : <u>_n \uparrow\downarrow d} \qquad \frac{c_i : u \uparrow\downarrow d \quad \forall i \in [1,n]}{\texttt{AnyOf} <c_1\ c_2\ \dots\ c_n> : u \uparrow\downarrow d}$$

3.5.3 Priority combinators

The last subset of combinators we introduce are *priority* combinators. In *BIP*, priority is used to prevent some interactions to be executed when other interactions are possible. Only interactions with maximal priority amongst the enabled interactions can ever be executed. The two following combinators introduce this notion to our theory of connectors.

- `FirstOf` introduces choice with priority. Interactions from the first underlying connector will be favoured to interactions from the second connector.
- `Maximal` is more general. Given a (partial) ordering, this connector returns all interactions from the underlying connector whose upwards values are maximal amongst the upwards values. Values are considered maximal if there does not exist a value which is strictly larger. The partial ordering $<$ will be encoded using a function f. For each pair (a, b), we will have that:

$$f(a,b) = 1 \iff a < b$$

Grammar

```
<PriorityC> ::= FirstOf <C> <C> | Maximal <f> <C>
```

for $f \in \mathcal{V}$.

Typing rules

$$\frac{c_1 : u \uparrow\downarrow d \qquad c_2 : u \uparrow\downarrow d}{\texttt{FirstOf}\ c_1\ c_2 : u \uparrow\downarrow d} \qquad \frac{f : (u \times u) \to \texttt{Bool} \qquad c : u \uparrow\downarrow d}{\texttt{Maximal}\ f\ c : u \uparrow\downarrow d}$$

Semantics

The semantics of those two connector combinators is given by:

$$[\texttt{FirstOf}\ c_1\ c_2](S) = \begin{cases} [c_1](S) & \text{if } [c_1](S) \neq \emptyset \\ [c_2](S) & \text{otherwise} \end{cases}$$
$$[\texttt{Maximal}\ f\ c](S) = \{(u, g) \mid (u, g) \in [c](S) \wedge \nexists (v, h) \in [c](S) . f(u, v) = 1\}$$

As you would expect, those two combinators can only filter out interactions, but not create nor modify interactions.

Derivation of `FirstOf`

Note that `FirstOf` is defined as a primitive combinator here. However, it could be derived from `Maximal`, `OneOf` and `Mapped`. The idea being to:

- Tag a priority to the upwards values of the two connectors using `Mapped`,
- Combine the two resulting combinators using `OneOf`,
- Apply `Maximal` with a function looking at the priority tags and,
- Finally, get rid of the tags using `Mapped` once again.

More formally, `FirstOf` could be defined as follows:

$$\texttt{FirstOf}\ c_1\ c_2 = \texttt{Mapped}\ first\ (\texttt{Maximal}\ compare_2\ (\texttt{OneOf}\ (\texttt{Mapped}\ tag_2\ c_1)\ (\texttt{Mapped}\ tag_1\ c_2)))$$

Where the various function are defined as:

$$first(a, b) = a$$
$$compare_2(a_1, b_1)(a_2, b_2) = \begin{cases} 1 & \text{if } b_1 < b_2 \\ 0 & \text{otherwise} \end{cases}$$
$$tag_n(x) = (x, n)$$

We decided to keep `FirstOf` as a primitive connector because it is very common and some interesting properties can be easily derived from it. Using the above construction would force us to inspect the partial order, which is cumbersome and might limit us in the implementation.

3.5.4 Dynamic combinators

The last set of connectors we introduce is the set of dynamic combinators. As the connector of a system is fixed once and for all before the system actually starts executing, we must have a way to involve in interactions atoms that are spawned later in the lifetime of the system. The two following combinators, `Dynamic` and `Joined`, allow us to perform exactly this, each in a specific and complementary manner.

- `Dynamic`, given a port p, connects this specific ports to all atoms, present or future. It can be thought as the disjunction of `Bind` connectors over the set of atoms $\mathcal{A}$.

  $$\texttt{Dynamic}\ p \approx \texttt{AnyOf}\ <\texttt{Bind}\ a_i\ p \mid a_i \in \mathcal{A}>$$

  The above notation is only valid if the set of atoms $\mathcal{A}$ is finite, as we have only defined `AnyOf` on finite sequences. Unfortunately, it is not always the case that the set of atoms is finite. Indeed, we considered the set of atoms to be possibly countably infinite to represent the fact that an unbounded number of atoms may be spawned during the life cycle of a system. For this reason, and to avoid dealing with infinite connectors, we introduce `Dynamic` as a primitive combinator.

- `Joined` is a particular kind of connector which blurs the line between the connectors and the values. Given a connectors c whose upwards values are themselves connectors, `Joined` c will in some sense behave as the connectors contained in upwards values. The name `Joined` was chosen because this combinator, as we will later see, corresponds to the natural transformation μ, called *join* or *multiply*, in the context of category theory and to the `join` function in Haskell. This combinator, as its typing rule will make evident, collapses two levels of connector types into a single one.

Typing rules

$$\frac{}{\texttt{Dynamic}\ p : \mathit{sendType}(p) \uparrow\downarrow \mathit{receiveType}(p)}$$

$$\frac{c : (u \uparrow\downarrow d) \uparrow\downarrow d}{\texttt{Joined}\ c : u \uparrow\downarrow d}$$

Semantics

$$\begin{aligned}
[\texttt{Dynamic}\ p](S) &= \{(S(a,p), \{x \mapsto \{(a,p) \mapsto x\} \mid x \in \mathit{sem}(\mathit{receiveType}(p))\}) \mid (a,p) \in \mathit{domain}(S)\} \\
[\texttt{Joined}\ c](S) &= \{(u_2, \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\}) \\
&\quad \mid (u_1, g_1) \in [c](S) \\
&\quad \wedge (u_2, g_2) \in [u_1](S) \\
&\quad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\}
\end{aligned}$$

Remark. The introduction of `Joined`, which is a very powerful combinator, makes it possible to derive some of the combinators we have introduced as primitives. In particular `BothOf` and `Guarded` can both be derived from a smaller set of primitive combinators and `Joined` as follows:

$$\begin{aligned}
\texttt{BothOf}\ c1\ c2 &:= \texttt{Joined}\ (\texttt{Mapped}\ \mathit{mapTag}_{c2}\ c1) \\
\mathit{mapTag}_c(x) &:= \texttt{Mapped}\ \mathit{tag}_x\ c \\
\mathit{tag}_x(y) &:= (x, y)
\end{aligned}$$

$$\begin{aligned}
\texttt{Guarded}\ f\ c &:= \texttt{Joined}\ (\texttt{Mapped}\ \mathit{ensure}_f\ c) \\
\mathit{ensure}_f(x) &:= \begin{cases} \texttt{Success}\ x & \text{if } f(x) = 1 \\ \texttt{Failure} & \text{otherwise} \end{cases}
\end{aligned}$$

3.5.5 Other derived combinators

Many useful connector combinators may be derived using the primitive combinators we have defined throughout this section. We present here some derived combinators that we think are of particular interest.

The `Not` combinator

The derived combinator `Not` takes a connector as parameter and returns an interaction if and only if the underlying connector does not return any interactions. It can be defined as:

$$\texttt{Not}\ c := \texttt{Guarded}\ \mathit{identity}\ (\texttt{FirstOf}\ (\texttt{Mapped}\ \mathit{never}\ c)\ (\texttt{Success}\ 1))$$

Where *identity* is the identity function and *never* is the function that always returns 0.

If in a given system state the connector c provides at least one interaction, then by construction all interactions returned by `FirstOf` (`Mapped` *never* c) (`Success` 1) will have 0 as upwards value, as the first branch of `FirstOf` is taken, and thus will not pass the `Guarded` *identity* combinator.

However, if the connector c doesn't provide any interactions, the right branch of `FirstOf`, namely `Success` 1, will be taken and a single interaction with upwards value 1 will be emitted. As this interaction satisfies the `Guarded` *identity* combinator, the interaction is propagated by the `Not` c connector.

Note that no interactions of c may escape `Not` c. If an interaction is produced by `Not` c, it will not involve any atom, meaning that the downwards function of the interaction will always return an empty assignment.

The `Witness` combinator

The `Witness` combinator provides a single interaction if its underlying connector provides a non-zero number of interactions. If the underlying connector c does not produce any interaction, then `Witness` c itself will not provide any interaction. This combinator can be defined as:

$$\texttt{Witness}\ c := \texttt{Not}\ (\texttt{Not}\ c)$$

An interesting property of this combinator is that the downwards function of `Witness` c, if any, will always produce empty assignments. For this reason, `Witness` c allows to check whether a connector is enabled or not without involving any atom that may be contained in the connector c.

3.6 Properties

3.6.1 Soundness

The first property we will prove is that if a connector is well-typed, then all operations performed by the semantics function are well-defined. The problematic operations are function applications and function compositions. Indeed, functions must only be fed values from their domain.

Lemma 1. *Given a connector c such that $c : u \uparrow\downarrow d$ and a system state $S \in \mathcal{S}$ then $x \in sem(u)$ for each $(x, f) \in [c](S)$.*

Proof. We prove the above lemma by structural induction on the typing rules.

Case `Bind` a p

If $c =$ `Bind` a $p \in P$, then $p : u \uparrow\downarrow d$ and, by definition, if $(a, p) \in domain(S)$, then:

$$S(a, p) \in sem(sendType(p)) = sem(u)$$

And thus the upward value is indeed of the right type. If $(a, p) \notin domain(S)$, then the proposition trivially holds, as [`Bind` a p]$(S) = \emptyset$.

Case `Success`

If $c =$ `Success` x, then, by the typing rule of `Success`, it must be the case that $x : u$ and thus that $x \in sem(u)$. The proposition thus trivially holds.

Case `Failure`

If $c =$ `Failure`, then $[c](S) = \emptyset$, thus the proposition trivially holds.

Case `OneOf`

If $c =$ `OneOf` c_1 c_2 then, by the typing rule of `OneOf`, it must be the case that $c_1 : u \uparrow\downarrow d$ and $c_2 : u \uparrow\downarrow d$. By induction hypothesis, it must be the case that the proposition holds for c_1 and c_2. Thus, the proposition holds for c, as $[c](S) = [c_1](S) \cup [c_2](S)$.

Case `BothOf`

If $c =$ `BothOf` c_1 c_2 then, by the typing rule of `BothOf` it must be the case that:

- $c_1 : v \uparrow\downarrow d$ for some type v.
- $c_2 : w \uparrow\downarrow d$ for some type w.
- $u = v \times w$ for the types v, w just introduced.

By induction hypothesis, the proposition holds for c_1 and c_2. Thus, we know that the proposition holds for c, as the upwards values of $[c](S)$ are pairs of upwards values from respectively $[c_1](S)$ and $[c_2](S)$.

Case `Mapped`

If $c =$ `Mapped` f c_1, then by the typing rule of `Mapped`, we have that:

- $f : v \to u$, for some type v. Thus, we have that $f \in \mathit{sem}(v) \to \mathit{sem}(u)$.
- $c_1 : v \uparrow\downarrow d$, for the type v introduced above.

By induction hypothesis, we know that the upwards values in $[c_1](S)$ are elements of $\mathit{sem}(v)$. Thus, the application of f to the upwards values of $[c_1](S)$ is well-defined and results in elements of $\mathit{sem}(u)$. The proposition thus holds for $c =$ `Mapped` f c_1.

Case `ContraMapped`

If $c =$ `ContraMapped` f c_1, then by the typing rule of `ContraMapped`, it must be the case that $c_1 : u \uparrow\downarrow e$ for some type e. The upwards values of $[c_1](S)$ are thus part of $\mathit{sem}(u)$. Therefore, the proposition trivially holds for $c =$ `ContraMapped` f c_1.

Case `Guarded`

If $c =$ `Guarded` f c_1, by the typing rule of `Guarded`, we have that:

- $f : u \to$ `Bool`, and thus $f \in \mathit{sem}(u) \to \{0, 1\}$.
- $c_1 : u \uparrow\downarrow d$.

By induction hypothesis, the upwards values of $[c_1](S)$ are elements of $\mathit{sem}(u)$. Therefore, their application to the function f is well defined. Moreover, as upwards values from [`Guarded` f c_1](S) are a subset of the upwards values of $[c_1](S)$, they also are a subset of $\mathit{sem}(u)$. The proposition thus holds for $c =$ `Guarded` f c_1.

Case `Feedback`

If $c =$ `Feedback` c_1 then the proposition hold trivially by induction hypothesis.

Case `FirstOf`

If $c =$ `FirstOf` c_1 c_2 then, by the typing rule of `FirstOf`, it must be the case that $c_1 : u \uparrow\downarrow d$ and $c_2 : u \uparrow\downarrow d$. By induction hypothesis, it must be the case that the proposition holds for c_1 and c_2. Thus, the proposition trivially holds for c.

Case `Maximal`

If $c =$ `Maximal` f c_1, then, by the typing rule of `Maximal`, we must have that:

- $f : (u \times u) \to$ `Bool`, and thus that $f \in (\mathit{sem}(u) \times \mathit{sem}(u)) \to \{0, 1\}$.
- $c_1 : u \uparrow\downarrow d$.

By induction hypothesis, we know that the proposition holds for c_1. Thus, the upwards values of $[c_1](S)$ are elements of $\mathit{sem}(u)$, and therefore so are the upwards values of $[c](S)$. In addition, the application of pairs of upwards values to f is thus well-defined. The proposition therefore holds for $c =$ `Maximal` f c_1.

Case `Dynamic`

Trivially, by construction, the upwards values of $c =$ `Dynamic` p will be of type $u = \mathit{sendType}(p)$.

Case `Joined`

If $c =$ `Joined` c_1 then, by the typing rule of `Joined`, it must be the case that $c_1 : (u \uparrow\downarrow d) \uparrow\downarrow d$. Thus, by induction hypothesis, it must be the case that upwards values of $[c_1](S)$ are connectors of type $u \uparrow\downarrow d$. Then, again by induction, it must be the case that upwards values of $[c_1](S)$, which are also the upwards values of $[c](S)$, are elements of $\mathit{sem}(u)$. Thus the proposition also holds for $c =$ `Joined` c_1.

This concludes the proof of the lemma. □

Lemma 2. *Given a connector c such that $c : u \uparrow\downarrow d$ and a system state $S \in \mathcal{S}$ then $sem(d) \subseteq domain(f)$ for each $(x, f) \in [c](S)$.*

Proof. We prove the above lemma by structural induction on the typing rules.

Case `Bind` $a\ p$

Let's consider $c = \texttt{Bind}\ a\ p$. If $(a, p) \in domain(S)$, then the domain of the downwards function is by definition $sem(receiveType(p))$. If $(a, p) \notin domain(S)$, then the proposition trivially holds, as $[\texttt{Bind}\ a\ p](S) = \emptyset$.

Case `Success`

If $c = \texttt{Success}\ x$, then by definition the downwards functions of $[c](S)$ are defined on the whole set of values $\mathcal{V}$, and thus the proposition trivially holds.

Case `Failure`

If $c = \texttt{Failure}$, then $[c](S) = \emptyset$, thus the proposition trivially holds.

Case `OneOf`

If $c = \texttt{OneOf}\ c_1\ c_2$ then, by the typing rule of `OneOf`, it must be the case that $c_1 : u \uparrow\downarrow d$ and $c_2 : u \uparrow\downarrow d$. By induction hypothesis, it must be the case that the proposition holds for c_1 and c_2. Thus, the proposition holds for c, as $[c](S) = [c_1](S) \cup [c_2](S)$.

Case `BothOf`

If $c = \texttt{BothOf}\ c_1\ c_2$ then, by the typing rule of `BothOf`, it must be the case that:

- $c_1 : v \uparrow\downarrow d$ for some type v.
- $c_2 : w \uparrow\downarrow d$ for some type w.

By induction hypothesis, the proposition holds for c_1 and c_2. Thus, we have that the domains of the downwards functions of $[c_1](S)$ and $[c_2](S)$ are subsets of $sem(d)$. Thus, the domains of the downwards functions of $[\texttt{BothOf}\ c_1\ c_2](S)$ are also subsets of $sem(d)$, as the intersection of two subsets is also a subset.

Case `Mapped`

By induction hypothesis, the proposition holds for c as the downwards functions are left unchanged.

Case `ContraMapped`

If $c = \texttt{ContraMapped}\ f\ c_1$, then by the typing rule of `ContraMapped`, we have that:

- $f : d \to e$, for some type e. Thus, we have that $f \in sem(d) \to sem(e)$ and thus $domain(f) = sem(d)$.
- $c_1 : u \uparrow\downarrow e$, for the type e introduced above.

By induction hypothesis, we know that $sem(e) \subseteq domain(g)$, for any downwards function g from $[c_1](S)$. Thus, as $range(f) \subseteq domain(g)$, the compositions $g \circ f$ are well-defined. As $domain(g \circ f) = domain(f) = sem(d)$, the proposition holds for $c =$ `ContraMapped` f c_1.

Case `Guarded`
Trivial by induction hypothesis, as downwards functions are left unchanged.

Case `Feedback`
If $c =$ `Feedback` c_1 then by the typing rule of `Feedback` it must be the case that c_1 is of type $u \uparrow\downarrow (d, u)$. By induction hypothesis, the downwards functions g from $[c_1](S)$ are all defined on $sem(d) \times sem(u)$. By the previous lemma, we also know that the upwards values x are all elements of $sem(u)$. Therefore, the composition of g with the tag_x function, $g \circ tag_x$, is well defined and has domain $\supseteq sem(d)$.

Case `FirstOf`
If $c =$ `FirstOf` c_1 c_2 then, by the typing rule of `FirstOf`, it must be the case that $c_1 : u \uparrow\downarrow d$ and $c_2 : u \uparrow\downarrow d$. By induction hypothesis, it must be the case that the proposition holds for c_1 and c_2. Thus, the proposition trivially holds for c.

Case `Maximal`
Trivial by induction hypothesis, as downwards functions are left unchanged.

Case `Dynamic` p
If $c =$ `Dynamic` p then by construction the domain of the downwards function is defined to be $sem(receiveType(p))$. Thus the proposition trivially holds for $c =$ `Dynamic` p.

Case `Joined`
If $c =$ `Joined` c_1 then, by the typing rule of `Joined`, it must be the case that $c_1 : (u \uparrow\downarrow d) \uparrow\downarrow d$. Thus, by the previous lemma, it must be the case that upwards values of $[c_1](S)$ are connectors of type $u \uparrow\downarrow d$. Moreover, by induction hypothesis, it must be the case that the downwards functions g_1 of $[c](S)$ are defined on all elements of $sem(d)$.

By induction, it must be the case that the property holds on downwards functions g_2 of $[c_1](S)$. That is, all g_2 are defined on all values of $sem(d)$. Thus, the intersection of the domains of g_1 and g_2 is itself a subset of $sem(d)$. Thus the proposition also holds for $c =$ `Joined` c_1.

This concludes the proof of the lemma. □

Theorem 1. *Given a connector c, such that $c : u \uparrow\downarrow u$ for some type u, then $[\![c]\!]$ is well-defined.*

Proof. Recall that, for any system state $S \in \mathcal{S}$, we have:

$$[\![c]\!](S) = \{close(o) \mid o \in [c](S)\} = \{g(x) \mid (x, g) \in [c](S)\}$$

From lemma 1 we know that upwards values of $[c](S)$ will be elements of $sem(u)$. Additionally, we know from lemma 2 that the each element of $sem(u)$ is part of the domain of every downwards function of $[c](S)$. If follows immediately that the application of an upwards value to their associated downwards function is well-defined, and thus $close(\cdot)$ and $[\![\cdot]\!]$ are also well-defined. This concludes the proof of the theorem. □

We will now show properties regarding how values are sent and received by the ports. The first property we will prove states that atom-port pairs can only receive a value if they are active, that is if a value was sent through the port by the atom.

Theorem 2. *Given a connector c, such that $c : u \uparrow\downarrow u$ for some type u, and a system state $S \in \mathcal{S}$, then, for each assignment $R \in [\![c]\!](S)$ we have that $domain(R) \subseteq domain(S)$. Intuitively, this means that atoms do no receive values on ports on which they didn't send any value.*

Proof. The above theorem is a straightforward consequence of the construction of the semantics function. The theorem follows directly from the fact that only in the case of `Bind` and `Dynamic` can new points be assigned to the domain of an assignment. Indeed, all other combinators either leave the assignments untouched or simply combine them. In the `Bind` and `Dynamic` cases, new atom-port pairs are only added to the domain of the assignment if the pair is part of the domain of the system state.

Thus, when a atom-port pair (a, p) is part of the domain of an assignment $R \in [\![c]\!](S)$, it must be, by construction, the case that $(a, p) \in domain(S)$. □

We will also show that assignments resulting from the application of a downwards function to two different downwards values have the same domain. That is to say, downwards values have no influence over whether or not an atom-port pair is assigned a value. They can, of course, potentially influence which values are received.

Theorem 3. *Given a connector c, such that $c : u \uparrow\downarrow d$ for some types u and d, a system state $S \in \mathcal{S}$ and any $(x, g) \in [c](S)$, and given two potentially different downwards values $y_1, y_2 \in sem(d)$, we have that $domain(g(y_1)) = domain(g(y_2))$.*

Proof. This theorem follows trivially by construction of the semantics function. Indeed, the downwards value is never inspected by any combinator. For all combinators, downwards values are in some sense indistinguishable. Not a single combinator can apply a different treatment to different downwards functions. □

Thus far, we have proven safety properties of the connector combinators. In the following section, we will investigate the complexity class of the semantics function.

3.6.2 Algebraic properties

In this section, we present some of the different algebraic laws that the connector combinators obey. These algebraic properties are of particular interest to us for many reasons:

- First of all, they highlight the abstract structure underlying the connector combinators. This view of the connector combinators should thus be very familiar to most mathematically-inclined readers.
- In addition, those algebraic properties will allow us to modify the structure of connectors while preserving the semantics. This will prove extremely useful for optimisation purposes.
- Finally, many of the algebraic concepts we will cover are present in Haskell in the form of type classes. Proving the algebraic properties on connector combinators $\mathcal{C}$ will provide justifications for the type class instances of the connectors in Haskell.

In the rest of this section, we will describe and prove a series of algebraic properties of connectors. But first, let's start by defining the equality of connectors.

Definition 1 (Connector equality)**.** We consider two connectors $c_1, c_2 \in \mathcal{C}$ to be equal if they have the same semantic value for any system state.

$$c_1 = c_2 \iff \forall S \in \mathcal{S}.[c_1](S) = [c_2](S)$$

Lemma 3. *Connector equality is* undecidable*, meaning that there exist no algorithm to decide in a finite amount of time whether or not two connectors are equal.*

Proof. The undecidability of connector equality derives directly from the undecidability of equality on functions in our language, which itself follows directly from Rice's theorem[16], since computable functions are a subset of our values $\mathcal{V}$. Rice's theorem states that any non-trivial property[1] on (partial) computable function is undecidable.

Towards a contradiction, assume that equality on connectors was decidable. We would then be able to decide whether the following two connectors are equal, for some functions $f, g \in \mathcal{V}$, port $p \in P$ and atom $a \in \mathcal{A}$:

$$\texttt{Mapped}\ f\ (\texttt{Bind}\ a\ p) = \texttt{Mapped}\ g\ (\texttt{Bind}\ a\ p)$$

Trivially, the above statement reduces to the following equality on functions in $\mathcal{V}$:

$$f = g$$

As being equal to f is a non-trivial property, meaning that there are functions equal and function not equal to f, deciding whether g is equal to f in a finite time is not possible according to Rice's theorem. Therefore we reach a contradiction, and thus it must be the case that connector equality is undecidable. □

[1] A non-trivial trivial property is defined as a property which holds for at least one, but not all, objects.

Remark. As we have just shown, deciding whether or not two connectors are equal is not always possible. We will however, in the rest of this section, discuss when certain classes of connectors are equal through the discussion of some algebraic properties of connectors.

Lemma 4. *($\mathcal{C}$, `OneOf`, `Failure`) is a commutative monoid, that is:*

- *`OneOf` is associative.*
- *`Failure` is the identity element of `OneOf`.*
- *`OneOf` is commutative.*

Proof. We prove the different properties of the monoid separately.

Associativity By definition, we have that:

$$\texttt{OneOf}\ c_1\ (\texttt{OneOf}\ c_2\ c_3) = \texttt{OneOf}\ (\texttt{OneOf}\ c_1\ c_2)\ c_3$$
$$\iff$$
$$\forall S \in \mathcal{S}.[\texttt{OneOf}\ c_1\ (\texttt{OneOf}\ c_2\ c_3)](S) = [\texttt{OneOf}\ (\texttt{OneOf}\ c_1\ c_2)\ c_3](S)$$

By associativity of the union of sets, the proposition follows immediately. Given any system state S, we have that:

$$\begin{aligned}
[\texttt{OneOf}\ c_1\ (\texttt{OneOf}\ c_2\ c_3)](S) &= [c_1](S) \cup [\texttt{OneOf}\ c_2\ c_3)](S) \\
&= [c_1](S) \cup ([c_2](S) \cup [c_3](S)) \\
&= ([c_1](S) \cup [c_2](S)) \cup [c_3](S) \\
&= [\texttt{OneOf}\ c_1\ c_2](S) \cup [c_3](S) \\
&= [\texttt{OneOf}\ (\texttt{OneOf}\ c_1\ c_2)\ c_3](S)
\end{aligned}$$

Identity element We must prove that, for any $c \in \mathcal{C}$:

$$\texttt{OneOf}\ c\ \texttt{Failure} = \texttt{OneOf}\ \texttt{Failure}\ c = c$$

By definition, this statement is equivalent for any system state S to:

$$[\texttt{OneOf}\ c\ \texttt{Failure}](S) = [\texttt{OneOf}\ \texttt{Failure}\ c](S) = [c](S)$$

The property follows directly then from the fact that the empty set is the identity element of set union:

$$[\texttt{OneOf}\ c\ \texttt{Failure}](S) = [c](S) \cup \emptyset = [c](S)$$
$$[\texttt{OneOf}\ \texttt{Failure}\ c](S) = \emptyset \cup [c](S) = [c](S)$$

Commutativity The commutativity of `OneOf` follows directly from commutativity of set union. By definition, we have that:

$$\texttt{OneOf}\ c_1\ c_2 = \texttt{OneOf}\ c_2\ c_1$$
$$\iff$$
$$\forall S \in \mathcal{S}.[\texttt{OneOf}\ c_1\ c_2](S) = [\texttt{OneOf}\ c_2\ c_1](S)$$

Moreover, we have that, for any system state S:

$$\begin{aligned}[\texttt{OneOf}\ c_1\ c_2](S) &= [c_1](S) \cup [c_2](S) \\ &= [c_2](S) \cup [c_1](S) \\ &= [\texttt{OneOf}\ c_2\ c_1](S)\end{aligned}$$

Thus, $(\mathcal{C}, \texttt{OneOf}, \texttt{Failure})$ is a commutative monoid. $\square$

Lemma 5. $(\mathcal{C}, \texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ \cdot\ \cdot), \texttt{Success}\ 1)$ *is a commutative monoid, given that* $\times$ *is the binary operation from a commutative monoid and* 1 *is its identity element. Precisely, this means that:*

- $\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ \cdot\ \cdot)$ *is associative.*
- $\texttt{Success}\ 1$ *is the identity element of* $\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ \cdot\ \cdot)$.
- $\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ \cdot\ \cdot)$ *is commutative.*

Proof. Let's prove the different properties one by one.

Associativity We must show that, for any $c_1, c_2, c_3 \in \mathcal{C}$, we have:

$$\begin{gathered}\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ (\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_2\ c_3)))\\=\\\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ (\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ c_2))\ c_3)\end{gathered}$$

By definition, the above statement is equivalent for any system state S to:

$$\begin{gathered}[\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ (\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_2\ c_3)))](S)\\=\\{}[\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ (\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ c_2))\ c_3)](S)\end{gathered}$$

The above equality holds, as will show the following derivation:

$$\begin{aligned}&[\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ (\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_2\ c_3)))](S)\\&= \{(x_1 \times y, \{x \mapsto g_1(x) \cup h(x) \mid x \in domain(g_1) \cap domain(h)\})\\&\quad \mid (x_1, g_1) \in [c_1](S)\\&\quad \wedge (y, h) \in [\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_2\ c_3)](S)\\&\quad \wedge g_1 \text{ and } h \text{ are downwards compatible}\}\\&= \{(x_1 \times x_2 \times x_3, \{x \mapsto g_1(x) \cup g_2(x) \cup g_3(x) \mid x \in \bigcap_{i=1}^{3} domain(g_i)\})\\&\quad \mid (x_1, g_1) \in [c_1](S)\\&\quad \wedge (x_2, g_2) \in [c_2](S)\\&\quad \wedge (x_3, g_3) \in [c_3](S)\\&\quad \wedge g_1, g_2 \text{ and } g_3 \text{ are downwards compatible}\}\\&= \{(z \times x_3, \{x \mapsto f(x) \cup g_3(x) \mid x \in domain(f) \cap domain(g_3)\})\\&\quad \mid (z, f) \in [\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ c_2)](S)\\&\quad \wedge (x_3, g_3) \in [c_3](S)\\&\quad \wedge f \text{ and } g_3 \text{ are downwards compatible}\}\\&= [\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ (\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ c_2))\ c_3)](S)\end{aligned}$$

Identity element We then show that `Success` 1 is the identity element of:

$$\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ \cdot\ \cdot)$$

That is, for all $c \in \mathcal{C}$ of the appropriate type, we have:

$$\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ (\texttt{Success}\ 1)\ c) = \texttt{Mapped}\ (\times)\ c\ (\texttt{BothOf}\ (\texttt{Success}\ 1)) = c$$

This follows directly from the fact that, for any system state S:

$$[\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ (\texttt{Success}\ 1)\ c)](S) = \{(x \times 1, g) \mid (x, g) \in [c](S)\} = [c](S)$$
$$[\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c\ (\texttt{Success}\ 1))](S) = \{(1 \times x, g) \mid (x, g) \in [c](S)\} = [c](S)$$

Commutativity The commutativity property of `Mapped` $(\times)$ (`BothOf` $\cdot\ \cdot$) states that, for any $c_1, c_2 \in \mathcal{C}$ with appropriate types, we have that:

$$\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ c_2) = \texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_2\ c_1)$$

The above statement is by definition equivalent to, for any system state S:

$$[\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ c_2)](S) = [\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_2\ c_1)](S)$$

The above proposition holds, as:

$$\begin{aligned}
[\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_1\ c_2)](S) &= \{(x_1 \times x_2, \{x \mapsto g_1(x) \cup g_2(x) \mid x \in \bigcap_{i=1}^{2} domain(g_i)\}) \\
&\quad \mid (x_1, g_1) \in [c_1](S) \\
&\quad \wedge (x_2, g_2) \in [c_2](S) \\
&\quad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\} \\
&= \{(x_2 \times x_1, \{x \mapsto g_2(x) \cup g_1(x) \mid x \in \bigcap_{i=1}^{2} domain(g_i)\}) \\
&\quad \mid (x_2, g_2) \in [c_1](S) \\
&\quad \wedge (x_1, g_1) \in [c_2](S) \\
&\quad \wedge g_2 \text{ and } g_1 \text{ are downwards compatible}\} \\
&= [\texttt{Mapped}\ (\times)\ (\texttt{BothOf}\ c_2\ c_1)](S)
\end{aligned}$$

Therefore, ($\mathcal{C}$, `Mapped` $(\times)$ (`BothOf` $\cdot\ \cdot$), `Success` 1) is a commutative monoid. □

Lemma 6. *BothOf distributes over OneOf, that is:*

$$\texttt{BothOf}\ c_1\ (\texttt{OneOf}\ c_2\ c_3) = \texttt{OneOf}\ (\texttt{BothOf}\ c_1\ c_2)\ (\texttt{BothOf}\ c_1\ c_3)$$

Proof. To prove the above lemma, we show that, for any connectors c_1, c_2, c_3 of appropriate types and for any system state S, the following holds:

$$[\texttt{BothOf}\ c_1\ (\texttt{OneOf}\ c_2\ c_3)](S) = [\texttt{OneOf}\ (\texttt{BothOf}\ c_1\ c_2)\ (\texttt{BothOf}\ c_1\ c_3)](S)$$

The following step-by-step derivation shows that in fact the previous statement holds.

$$
\begin{aligned}
&[\texttt{BothOf}\ c_1\ (\texttt{OneOf}\ c_2\ c_3)](S)\\
&\quad = \{(x_1 \times y, \{x \mapsto g_1(x) \cup h(x) \mid x \in domain(g_1) \cap domain(h))\})\\
&\qquad \mid (x_1, g_1) \in [c_1](S)\\
&\qquad \wedge (y, h) \in [\texttt{OneOf}\ c_2\ c_3](S)\\
&\qquad \wedge g_1 \text{ and } h \text{ are downwards compatible}\}\\
&\quad = \{(x_1 \times y, \{x \mapsto g_1(x) \cup h(x) \mid x \in domain(g_1) \cap domain(h))\})\\
&\qquad \mid (x_1, g_1) \in [c_1](S)\\
&\qquad \wedge (y, h) \in [c_2](S) \cup [c_3](S)\\
&\qquad \wedge g_1 \text{ and } h \text{ are downwards compatible}\}\\
&\quad = \{(x_1 \times x_2, \{x \mapsto g_1(x) \cup g_2(x) \mid x \in domain(g_1) \cap domain(g_2))\})\\
&\qquad \mid (x_1, g_1) \in [c_1](S)\\
&\qquad \wedge (x_2, g_2) \in [c_2](S)\\
&\qquad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\}\\
&\quad \cup\\
&\qquad \{(x_1 \times x_3, \{x \mapsto g_1(x) \cup g_3(x) \mid x \in domain(g_1) \cap domain(g_3))\})\\
&\qquad \mid (x_1, g_1) \in [c_1](S)\\
&\qquad \wedge (x_3, g_3) \in [c_3](S)\\
&\qquad \wedge g_1 \text{ and } g_3 \text{ are downwards compatible}\}\\
&\quad = [\texttt{BothOf}\ c_1\ c_2](S) \cup [\texttt{BothOf}\ c_1\ c_3](S)\\
&\quad = [\texttt{OneOf}\ (\texttt{BothOf}\ c_1\ c_2)\ (\texttt{BothOf}\ c_1\ c_3)](S)
\end{aligned}
$$

□

Lemma 7. *$\texttt{BothOf} \cdot \texttt{Failure}$ annihilates $\mathcal{C}$, that is for any connector $c \in \mathcal{C}$:*

$$\texttt{BothOf}\ c\ \texttt{Failure} = \texttt{Failure}$$

Proof. The above lemma follows trivially from the fact that the intersection with the empty set is always empty. □

Lemma 8. *$\texttt{OneOf}$ is idempotent, that is $\texttt{OneOf}\ c\ c = c$.*

Proof. The above lemma follows trivially from the fact that the set union is idempotent. □

Lemma 9. *$\texttt{Mapped}$ distributes over $\texttt{ContraMapped}$, $\texttt{OneOf}$ and $\texttt{FirstOf}$. That is, for any connectors $c, c_1, c_2 \in C$ and functions $f, g \in \mathcal{V}$ of the appropriate types, the following hold:*

$$\begin{aligned}
\texttt{Mapped}\ f\ (\texttt{ContraMapped}\ g\ c) &= \texttt{ContraMapped}\ g\ (\texttt{Mapped}\ f\ c)\\
\texttt{Mapped}\ f\ (\texttt{OneOf}\ c_1\ c_2) &= \texttt{OneOf}\ (\texttt{Mapped}\ f\ c_1)\ (\texttt{Mapped}\ f\ c_2)\\
\texttt{Mapped}\ f\ (\texttt{FirstOf}\ c_1\ c_2) &= \texttt{FirstOf}\ (\texttt{Mapped}\ f\ c_1)\ (\texttt{Mapped}\ f\ c_2)
\end{aligned}$$

Proof. The above statements follow trivially from the definition of the semantics. □

Lemma 10. *$\texttt{ContraMapped}$ distributes over $\texttt{Mapped}$, $\texttt{OneOf}$ and $\texttt{FirstOf}$. For any connectors $c, c_1, c_2 \in C$ and functions $f, g \in \mathcal{V}$ of the appropriate types, the following hold:*

$$\begin{aligned}
\texttt{ContraMapped}\ f\ (\texttt{Mapped}\ g\ c) &= \texttt{Mapped}\ g\ (\texttt{ContraMapped}\ f\ c)\\
\texttt{ContraMapped}\ f\ (\texttt{OneOf}\ c_1\ c_2) &= \texttt{OneOf}\ (\texttt{ContraMapped}\ f\ c_1)\ (\texttt{ContraMapped}\ f\ c_2)\\
\texttt{ContraMapped}\ f\ (\texttt{FirstOf}\ c_1\ c_2) &= \texttt{FirstOf}\ (\texttt{ContraMapped}\ f\ c_1)\ (\texttt{ContraMapped}\ f\ c_2)
\end{aligned}$$

Proof. As before, the above statements follow directly from the definition of the semantics. □

3.7 Properties from Category Theory

The algebraic structures in the next section are all rooted in Category Theory[17]. They are of particular interest since they are all present in Haskell in the form of type classes[18]. The proofs that connectors obey the various algebraic laws in this section will justify the implementation of the type class instances in Haskell and the various methods in Scala. Since notions from Category Theory might not be familiar to all readers, an appendix on the subject can be found at the end of this thesis.

Lemma 11. *The values $\mathcal{V}$ and types forms a category, whose objects are the types and whose arrows are functions $f : a \to b \in \mathcal{V}$ [2] for some types a, b. We will refer to this category simply by $\mathcal{V}$.*

Proof. We show that all properties required by a category are indeed respected by the above formulation of a category for the values and types.

Composition Let us be given two arrows $f : a \to b$ and $g : b \to c$. The composition of f and g, $g \circ f$ is itself, by construction, a member of $\mathcal{V}$ and has type $a \to c$. Therefore, the compositions of arrows is well-defined in the category. The associativity of arrow composition follows directly from associativity of function composition.

Identity By definition, the identity function $identity_a : a \to a$ exists for any type a. Those arrows trivially act as identity with respect to arrow composition.

Therefore, the above formulation is indeed a category. □

[2]Remark that the notations for function type signatures, arrow signatures, as well as function signatures, are the same. For the scope of this thesis, this will not pose any problem since the concepts all coincide. The context will make clear which of those concepts is meant.

Lemma 12. *For any type d, the mapping F_{up} that maps types a to connector types $F_{up}(a) = a \uparrow\downarrow d$ and that maps arrows $f : a \to b$ to $F_{up}(f) =$ `Mapped` $f : (a \uparrow\downarrow d) \to (b \uparrow\downarrow d)$ is a functor from the category $\mathcal{V}$ of values and types to itself.*

Proof. To show that F_{up} is a functor, we show that identity and composition are preserved.

Identity We have to prove that, for any type u and d, $F_{up}(1_u) = 1_{F_{up}(u)}$. Let u by any type. By definition, 1_u is the identity function $identity_u : u \to u$. Therefore, we have that:

$$F_{up}(1_u) = \texttt{Mapped}\ identity_u$$

We observe that $1_{F_{up}(u)}$ is the identity function that maps connectors of type $u \uparrow\downarrow d$ to themselves. Therefore, for the identity preservation property to hold, we must have that for any connector c of the correct type the following holds:

$$\texttt{Mapped}\ identity_u\ c = c$$

By definition, the above equality is equivalent to the following statement:

$$\forall S \in \mathcal{S}.[\texttt{Mapped}\ identity_u\ c](S) = [c](S)$$

Let S be any system state. We have that the above statement is trivially true as:

$$\begin{aligned}[\texttt{Mapped}\ identity_u\ c](S) &= \{(identity_u(x), g) \mid (x, g) \in [c](S)\} \\ &= \{(x, g) \mid (x, g) \in [c](S)\} \\ &= [c](S)\end{aligned}$$

Composition We are left to show that indeed $F_{up}(g \circ f) = F_{up}(g) \circ F_{up}(f)$. In this context of this particular functor, we must show that:

$$\texttt{Mapped}\ (g \circ f) = \texttt{Mapped}\ g \circ \texttt{Mapped}\ f$$

That is, for any connector c of appropriate type, we must have that:

$$\texttt{Mapped}\ (g \circ f)\ c = \texttt{Mapped}\ g\ (\texttt{Mapped}\ f\ c)$$

By definition, this equality reduces to:

$$\forall S \in \mathcal{S}.[\texttt{Mapped}\ (g \circ f)\ c](S) = [\texttt{Mapped}\ g\ (\texttt{Mapped}\ f\ c)](S)$$

Let S be any system state. We trivially have that:

$$\begin{aligned}[\texttt{Mapped}\ (g \circ f)\ c](S) &= \{(g(f(x)), h) \mid (x, h) \in [c](S)\} \\ &= \{(g(y), h) \mid (y, h) \in [\texttt{Mapped}\ f\ c](S)\} \\ &= [\texttt{Mapped}\ g\ (\texttt{Mapped}\ f\ c)](S)\end{aligned}$$

This concludes the proof of that F_{up} is indeed a covariant functor. □

Lemma 13. *For any type u, the mapping F_{down} that maps the types d to $F_{down}(d) = u \uparrow\downarrow d$ and that maps functions $f : a \to b \in \mathcal{V}$ to $F_{down}(f) =$ `ContraMapped` $f : u \uparrow\downarrow b \to u \uparrow\downarrow a$ is a contravariant functor from the category $\mathcal{V}$ of values and types to itself.*

Proof. To prove that F_{down} is in fact a contravariant functor, we show that identity is preserved and that composition is reversed.

Identity We have to show that, for any type u and d, $F_{down}(1_d) = 1_{F_{down}(d)}$. Let d by any type. By definition, 1_d is the identity function $identity_d : d \to d$. Thus, we have that:

$$F_{down}(1_d) = \texttt{ContraMapped}\ identity_d$$

We observe that $1_{F_{down}(d)}$ is the identity function that maps connectors of type $u \uparrow\downarrow d$ to themselves. Therefore, for the identity preservation property to hold, we must have that for any connector c of the appropriate type the following holds:

$$\texttt{ContraMapped}\ identity_d\ c = c$$

By definition, the above equality is equivalent to the following statement:

$$\forall S \in \mathcal{S}.[\texttt{ContraMapped}\ identity_d\ c](S) = [c](S)$$

Let S be any system state. We show that the above statement is indeed true as:

$$\begin{aligned}[\texttt{ContraMapped}\ identity_d\ c](S) &= \{(x, g \circ identity_d) \mid (x, g) \in [c](S)\} \\ &= \{(x, g) \mid (x, g) \in [c](S)\} \\ &= [c](S)\end{aligned}$$

Composition We are left to show that indeed composition is indeed reversed, that is $F_{down}(g \circ f) = F_{down}(f) \circ F_{down}(g)$. In this context of this particular contravariant functor, we must show that:

$$\texttt{ContraMapped}\ (g \circ f) = \texttt{ContraMapped}\ f \circ \texttt{ContraMapped}\ g$$

That is, for any connector c of appropriate type, we must have that:

$$\texttt{ContraMapped}\ (g \circ f)\ c = \texttt{ContraMapped}\ f\ (\texttt{ContraMapped}\ g\ c)$$

By definition of connector equality, the previous statement reduces to:

$$\forall S \in \mathcal{S}.[\texttt{ContraMapped}\ (g \circ f)\ c](S) = [\texttt{ContraMapped}\ f\ (\texttt{ContraMapped}\ g\ c)](S)$$

Let S be any system state. We have that:

$$\begin{aligned}[\texttt{ContraMapped}\ (g \circ f)\ c](S) &= \{(x, h \circ g \circ f) \mid (x, h) \in [c](S)\} \\ &= \{(x, h \circ f) \mid (x, h) \in [\texttt{ContraMapped}\ g\ c](S)\} \\ &= [\texttt{ContraMapped}\ f\ (\texttt{ContraMapped}\ g\ c)](S)\end{aligned}$$

This concludes the proof of that F_{down} is indeed a contravariant functor. □

Lemma 14. *The family of mappings $\eta^{\mathcal{V}}$ that associates to each values $x \in \mathcal{V}$ the connector* `Success` *x is a natural transformation from $1_{\mathcal{V}}$ to F_{up}.*

Proof. We have to prove that the following equality holds for any function $f : a \to b \in \mathcal{V}$:

$$\eta_b^{\mathcal{V}} \circ 1_{\mathcal{V}}(f) = F_{up}(f) \circ \eta_a^{\mathcal{V}}$$

Which, in this context, translates to:

$$\texttt{Success} \circ f = \texttt{Mapped}\ f \circ \texttt{Success}$$

Or, for any value $x \in \mathcal{V}$:

$$\texttt{Success}\ f(x) = \texttt{Mapped}\ f\ (\texttt{Success}\ x)$$

Which is trivially true by definition of the semantics. □

Lemma 15. *The family of mappings $\mu^{\mathcal{V}}$ that associates to each connector $c \in \mathcal{C}$ of type $(u \uparrow\downarrow d) \uparrow\downarrow d$ for some types u, d, the connector* `Joined` c *is a natural transformation from $F_{up} \circ F_{up}$ to F_{up}.*

Proof. We have to prove that the following equality holds for any function $f : a \to b \in \mathcal{V}$:

$$\mu_b^{\mathcal{V}} \circ F_{up}(F_{up}(f)) = F_{up}(f) \circ \mu_a^{\mathcal{V}}$$

Which, in this context, translates to:

$$\texttt{Joined} \circ \texttt{Mapped}\ (\texttt{Mapped}\ f) = \texttt{Mapped}\ f \circ \texttt{Joined}$$

Or, for any value $c \in \mathcal{C}$ of type $(u \uparrow\downarrow d) \uparrow\downarrow d$:

$$\texttt{Joined}\ (\texttt{Mapped}\ (\texttt{Mapped}\ f)\ c) = \texttt{Mapped}\ f\ (\texttt{Joined}\ c)$$

By definition of the equality on connectors, the above equality is equivalent to the following statement:

$$\forall S \in \mathcal{S}.[\texttt{Joined}\ (\texttt{Mapped}\ (\texttt{Mapped}\ f)\ c)](S) = [\texttt{Mapped}\ f\ (\texttt{Joined}\ c)](S)$$

Let S be any system state. We have that:

$$\begin{aligned}
&[\texttt{Joined}\ (\texttt{Mapped}\ (\texttt{Mapped}\ f)\ c)](S)\\
&\quad= \{(u_2, \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\})\\
&\qquad \mid (u_1, g_1) \in [\texttt{Mapped}\ (\texttt{Mapped}\ f)\ c](S)\\
&\qquad \wedge (u_2, g_2) \in [u_1](S)\\
&\qquad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\}\\
&\quad= \{(u_2, \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\})\\
&\qquad \mid (u_1, g_1) \in [c](S)\\
&\qquad \wedge (u_2, g_2) \in [\texttt{Mapped}\ f\ u_1](S)\\
&\qquad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\}\\
&\quad= \{(f(u_2), \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\})\\
&\qquad \mid (u_1, g_1) \in [c](S)\\
&\qquad \wedge (u_2, g_2) \in [u_1](S)\\
&\qquad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\}\\
&\quad= [\texttt{Mapped}\ f\ (\texttt{Joined}\ c)](S)
\end{aligned}$$

This concludes the proof that $\mu^{\mathcal{V}}$ is indeed a natural transformation. □

Lemma 16. *The functor F_{up} along with natural transformations $\eta^{\mathcal{V}}$ and $\mu^{\mathcal{V}}$ form a monad.*

Proof. We have left to show that, for any value $x \in \mathcal{V}$, the following monad laws hold:

$$\mu^{\mathcal{V}}_x \circ F_{up}(\mu^{\mathcal{V}}_x) = \mu^{\mathcal{V}}_x \circ \mu^{\mathcal{V}}_x$$
$$\mu^{\mathcal{V}}_x \circ F_{up}(\eta^{\mathcal{V}}_x) = \mu^{\mathcal{V}}_x \circ \eta^{\mathcal{V}}_x = 1_x$$

Proof of first law The first law translates in this context to:

$$\texttt{Joined} \circ \texttt{Mapped Joined} = \texttt{Joined} \circ \texttt{Joined}$$

Which reduces, for any connector c of type $((u \uparrow\downarrow d) \uparrow\downarrow d) \uparrow\downarrow d$ and any system state S, to:

$$[\texttt{Joined}\ (\texttt{Mapped Joined}\ c)](S) = [\texttt{Joined}\ (\texttt{Joined}\ c)](S)$$

This equality can be derived as follows:

$$\begin{aligned}
&[\texttt{Joined}\ (\texttt{Mapped Joined}\ c)](S)\\
&\quad = \{(u_2, \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\})\\
&\qquad \mid (u_1, g_1) \in [\texttt{Mapped Joined}\ c](S)\\
&\qquad \wedge (u_2, g_2) \in [u_1](S)\\
&\qquad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\}\\
&\quad = \{(u_2, \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\})\\
&\qquad \mid (u_1, g_1) \in [c](S)\\
&\qquad \wedge (u_2, g_2) \in [\texttt{Joined}\ u_1](S)\\
&\qquad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\}\\
&\quad = \{(u_2, \{(x, g_1(x) \cup g_2(x) \cup g_3(x))\\
&\qquad\qquad \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2) \cap \mathit{domain}(g_3)\})\\
&\qquad \mid (u_1, g_1) \in [c](S)\\
&\qquad \wedge (u_3, g_3) \in [u_1](S)\\
&\qquad \wedge (u_2, g_2) \in [u_3](S)\\
&\qquad \wedge g_1, g_2 \text{ and } g_3 \text{ are downwards compatible}\}\\
&\quad = \{(u_2, \{(x, g_2(x) \cup g_3(x)) \mid x \in \mathit{domain}(g_2) \cap \mathit{domain}(g_3)\})\\
&\qquad \mid (u_3, g_3) \in [\texttt{Joined}\ c](S)\\
&\qquad \wedge (u_2, g_2) \in [u_3](S)\\
&\qquad \wedge g_2 \text{ and } g_3 \text{ are downwards compatible}\}\\
&\quad = [\texttt{Joined}\ (\texttt{Joined}\ c)](S)
\end{aligned}$$

Which concludes the proof of the first law.

Proof of second law In this particular context, the second law translates to:

$$\texttt{Joined} \circ \texttt{Mapped Success} = \texttt{Joined} \circ \texttt{Success} = \mathit{identity}$$

For any connector $c \in \mathcal{C}$, this means that:

$$\begin{aligned}\texttt{Joined}\ (\texttt{Mapped Success}\ c) &= c \\ \texttt{Joined}\ (\texttt{Success}\ c) &= c\end{aligned}$$

Those statements are proven by the following derivations. Let S be any system state.

$$\begin{aligned}
&[\texttt{Joined}\ (\texttt{Mapped Success}\ c)](S) \\
&\qquad = \{(u_2, \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\}) \\
&\qquad\quad \mid (u_1, g_1) \in [\texttt{Mapped Success}\ c](S) \\
&\qquad\quad \wedge (u_2, g_2) \in [u_1](S) \\
&\qquad\quad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\} \\
&\qquad = \{(u_2, \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\}) \\
&\qquad\quad \mid (u_1, g_1) \in [c](S) \\
&\qquad\quad \wedge (u_2, g_2) \in [\texttt{Success}\ u_1](S) \\
&\qquad\quad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\} \\
&\qquad = \{(u_1, g_1)) \mid (u_1, g_1) \in [c](S)\} \\
&\qquad = [c](S)
\end{aligned}$$

$$\begin{aligned}
&[\texttt{Joined}\ (\texttt{Success}\ c)](S) \\
&\qquad = \{(u_2, \{(x, g_1(x) \cup g_2(x)) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\}) \\
&\qquad\quad \mid (u_1, g_1) \in [\texttt{Success}\ c](S) \\
&\qquad\quad \wedge (u_2, g_2) \in [u_1](S) \\
&\qquad\quad \wedge g_1 \text{ and } g_2 \text{ are downwards compatible}\} \\
&\qquad = \{(u_2, g_2) \mid (u_2, g_2) \in [c](S)\} \\
&\qquad = [c](S)
\end{aligned}$$

This conclude the proof of the monad laws. □

3.7.1 Hardness

In this section, we investigate the complexity of the semantics function. In particular, we show that deciding whether a connector and a given system state have a possible assignment according to the semantics is a hard problem. More precisely, we show that this problem, under certain restrictions, is *NP-Complete*, that is both *NP-hard* and *NP*.

Definition 2. `CONNECTOR-ENABLEDNESS` We define `CONNECTOR-ENABLEDNESS` to be the decision problem that, given as input a set of atoms $\mathcal{A}$, a set of ports P, a system state $S \in \mathcal{S}$ and well-typed connector $c \in \mathcal{C}$, returns `YES` if $[\![c]\!](S)$ is non-empty, `NO` otherwise.

Theorem 4. *`CONNECTOR-ENABLEDNESS` is NP-hard, even if the functions used within the connectors are all computable in polynomial time.*

Proof. We provide a polynomial-time reduction of `3-SAT`, which is a *NP-hard* problem, to `CONNECTOR-ENABLEDNESS`.

`3-SAT` For recall, `3-SAT` is the decision problem that, given a boolean formula in conjunctive normal form, where each clause contains 3 literals, returns `YES` if a satisfying assignment of a truth value to each variable exists, and `NO` otherwise.

`3-SAT` **input** Let us be given a `3-SAT` instance, which is a boolean formula ϕ in conjunctive normal form. The input thus consists of the conjunction of n disjunctions of each 3 literals. Each literal can appear either in positive or negative form.

$$\phi = \bigwedge_{i=1}^{n} (x_{i1} \vee x_{i2} \vee x_{i3})$$

We are given a function $sign(\cdot)$ which returns, for each x_{ij}, 0 if the literal is in negative form, and 1 if the literal is in positive form. We are also given a $variable(\cdot)$ function, which returns a number $n \in [1, 3n]$ for each x_{ij}. Two literals with the same number correspond thus to the same variable.

Transformation Given this `3-SAT` instance, we produce a `CONNECTOR-ENABLEDNESS` instance $(\mathcal{A}, P, S, c)$ as follows. First, lets define our set of n atoms $\mathcal{A}$ and 3 distinct ports P:

$$\mathcal{A} = \{a_i \mid i \in [1, n]\}$$
$$P = \{p_j \mid j \in [1, 3]\}$$

We have thus an atom-port pair $(a_i, p_j) \in \mathcal{A} \times P$ for each literal x_{ij}.

We then construct a system state $S \in \mathcal{S}$ as follows:

$$S = \{((a_i, p_j), (variable(x_{ij}), sign(x_{ij}))) \mid i \in [1, n] \wedge j \in [1, 3]\}$$

The state S contains for each atom-port pair the variable and sign of the corresponding `3-SAT` literal.

We then proceed to define the connector $c \in \mathcal{C}$. For each disjunction $(x_{i1} \vee x_{i2} \vee x_{i3})$, we define the connector $d_i \in C$ as follows:

$$d_i = \texttt{AnyOf} <\texttt{Bind}\ a_i\ p_1, \texttt{Bind}\ a_i\ p_2, \texttt{Bind}\ a_i\ p_3> \qquad \forall i \in [1, n]$$

We define the connector $c \in \mathcal{C}$ to be:

$$c = \texttt{Guarded}\ \mathit{consistent}\ (\texttt{AllOf}\ <d_1, d_2 \ldots d_n>)$$

For recall, `AllOf` is the n-ary equivalent to `BothOf`. This connector propagates upwards the sequence of underlying upwards values. The function *consistent* is defined as follows:

$$\mathit{consistent}(s) = \begin{cases} 0 & \text{if } \exists v, i, j \in [1, n].s(i) = (v, 0) \wedge s(j) = (v, 1) \\ 1 & \text{otherwise} \end{cases}$$

Given a sequence of size n containing pairs, the $\mathit{consistent}(\cdot)$ function returns 0 if there exist in the input sequence two pairs which agree on their first element but disagree on the second. This function trivially runs in polynomial time.

We have now completed the description of the transformation of a `3-SAT` instance to `CONNECTOR-ENABLEDNESS`. To complete this proof, we will need to show the following facts:

- The connector can be well-typed.
- The transformation can be done in polynomial time.
- `NO`-instances of `3-SAT` are transformed to `NO`-instances of `CONNECTOR-ENABLEDNESS`.
- `YES`-instances of `3-SAT` are transformed to `YES`-instances of `CONNECTOR-ENABLEDNESS`.

Typing The ports of the system will all be given the type $(\mathit{Int} \times \mathit{Bool}) \uparrow\downarrow <\mathit{Int} \times \mathit{Bool}>_n$. The consistent function is given the type $<\mathit{Int} \times \mathit{Bool}>_n \rightarrow \mathit{Bool}$. Trivially, the connector c has thus type $<\mathit{Int} \times \mathit{Bool}>_n \uparrow\downarrow <\mathit{Int} \times \mathit{Bool}>_n$ and therefore the $[\![\cdot]\!]$ function is defined on the connector c.

Polynomial-time transformation The first observation we make is that the size of the input of `3-SAT` is in the order of n, where, as we recall, n is the number of disjunctive clauses. The set of ports P and the system state S are both of size in the order of n and can be straightforwardly constructed in polynomial time. The connector c is composed of a polynomial number of combinators and, as the $\mathit{consistent}(\cdot)$ can also be encoded in polynomial time, c can also be constructed in polynomial time. Therefore, the transformation can be performed in polynomial time.

Correspondence of `NO`-instances Given a `NO`-instance of `3-SAT`, we show that the transformed instance is a `NO`-instance of `CONNECTOR-ENABLEDNESS`. We prove this statement by contradiction. Suppose that we are given an instance $(\mathcal{A}, P, S, c)$ corresponding to a `NO`-instance ϕ of `3-SAT`, where ϕ, P, S and c are defined as above. Towards a contradiction, assume that $[\![c]\!](S)$ is non-empty. We thus have an assignment $R \in [\![c]\!](S)$. The domain of the assignment R is the set of ports that take part of the interaction. By construction, it must be the case that, for each $i \in [1, n]$, a single atom-port (a_i, p_j) is part of the domain of R. Without loss of generality, as `OneOf` is commutative, lets consider that the first port p_1 is selected in each case, and thus that the domain of R consists of the atom-ports pairs (a_i, p_1) for $i \in [1, n]$. Also by construction, it must be the case that the upwards value of `AllOf` $<d_1, d_2 \ldots d_n>$ satisfies the *consistent*$(\cdot)$ function. Therefore, it must be the case that there is no x_{i1}, x_{j1} such that *variable*(x_{i1}) = *variable*(x_{i1}) but *sign*$(x_{i1}) \neq$ *sign*(x_{i1}). We have reached here a contradiction as we can trivially construct a satisfying assignment α for the formula ϕ. Therefore, $[\![c]\!](S)$ must be empty and thus the transformed instance must be a `NO`-instance of `CONNECTOR-ENABLEDNESS`.

Correspondence of `YES`-instances Given a `YES`-instance of `3-SAT`, we show that the transformed instance is a `YES`-instance of `CONNECTOR-ENABLEDNESS`. Suppose that we are given an instance $(\mathcal{A}, P, S, c)$ corresponding to a `YES`-instance ϕ of `3-SAT`, where ϕ, $\mathcal{A}$, P, S and c are defined as above. Let us be given a satisfying assignment α that assigns to each variable a truth value. As the formula ϕ is satisfied by α, we must have at least a true literal in $(x_{i1} \vee x_{i2} \vee x_{i3})$ for each i. Without loss of generality, as $\vee$ is commutative, lets consider that x_{i1} is a true literal for each i. Let us construct a sequence s of size n:

$$s(i) = (\mathit{variable}(x_{i1}), \mathit{sign}(x_{i1})) \qquad \forall i \in [1, n]$$

We observe that, as each literal x_{i1} is a true literal, it follows directly that *consistent*$(s) = 1$. Let us construct an assignment R as follows:

$$R = \{(a_i, p_1) \mapsto s \mid i \in [1, n]\}$$

We show that $R \in [\![c]\!](S)$. Trivially, when the port p_1 is selected within all the connectors d_i, s is an upwards value of $[c](S)$. Thus, we must have that the assignment R is in $[\![c]\!](S)$. Therefore, $[\![c]\!](S)$ is non-empty and thus the transformed instance is a `YES`-instance of `CONNECTOR-ENABLEDNESS`.

This concludes the proof of the theorem. □

Theorem 5. *`CONNECTOR-ENABLEDNESS` is in NP when the functions used within the connectors are all computable in polynomial time and when `Joined` is not part of the connectors.*

Proof. We prove the above theorem by showing the construction of a polynomial time verifier for `CONNECTOR-ENABLEDNESS`. The verifier takes as additional input a proof which consists of a single bit for each `OneOf` combinator part of the connector. For each `OneOf` combinator, a 0 in the proof indicates that the left underlying connector must be selected, and a 1 indicates that the right underlying connector should be selected instead. Also part of the proof is a number for each `Dynamic` combinator. This number indicates which of the atom $a \in \mathcal{A}$ must be selected.

Trivially, the size of the proof is polynomial in the size of the original input to the `CONNECTOR-ENABLEDNESS` problem, as the proof consists of a linear number of non-negative integers, whose size is bounded by either 1 in the case bits or by the size of the original input in the case of atoms.

The first step consists of eliminating all non-determinism from the connector, by replacing all `OneOf` $c_1 c_2$ in the connector by either c_1 or c_2 depending on the proof, and all `Dynamic` p by `Bind` a p, where the corresponding atom a is given by the proof. This transformation can be trivially performed in polynomial time.

Then, once all non-determinism as been eliminated, the semantic function is evaluated. This evaluation can be trivially performed in polynomial time, as a straightforward (and long!) case by case analysis of the different remaining combinators would show.

If the evaluation leads to a non-empty set of interactions (in this case exactly 1 interaction), then we have indeed been able to prove that indeed the `CONNECTOR-ENABLEDNESS` instance is a satisfying instance.

This concludes the proof that `CONNECTOR-ENABLEDNESS`, under the given restrictions, is in *NP*. □

3.7.2 Stability

In this section, we look at stability properties, that is properties that, when holding for a given system state, hold for all larger states. This type of properties will be useful when we look at the possibility of executing interactions even when there are atoms still executing in the system.

Definition 3 (State maximality)**.** Given a well-typed connector $c \in \mathcal{C}$, a state $S \in \mathcal{S}$ is *maximal* for c if and only if:

$$\forall S' \in \mathcal{S} . S \subseteq S' \Rightarrow [\![c]\!](S) = [\![c]\!](S')$$

Intuitively, a state is maximal if and only if activating new atom-port pairs does not change the set of possible interactions of the connector.

Theorem 6. *The given derivation rules for maximality hold for any state $S \in \mathcal{S}$, any interaction $\gamma \in \mathcal{O}$, any connectors $c, c_1, c_2 \dots \in \mathcal{C}$, any atom $a \in \mathcal{A}$, any port $p \in$ and any $f \in \mathcal{V}$ of the appropriate types:*

$$\frac{[\texttt{Bind}\ a\ p](S) \neq \emptyset}{S \text{ maximal for } \texttt{Bind}\ a\ p}$$

$$\frac{}{S \text{ maximal for } \texttt{Success}\ x} \qquad \frac{}{S \text{ maximal for } \texttt{Failure}}$$

$$\frac{S \text{ maximal for } c_1 \qquad S \text{ maximal for } c_2}{S \text{ maximal for } \texttt{OneOf}\ c_1\ c_2} \qquad \frac{S \text{ maximal for } c_1 \qquad S \text{ maximal for } c_2}{S \text{ maximal for } \texttt{BothOf}\ c_1\ c_2}$$

$$\frac{S \text{ maximal for } c}{S \text{ maximal for } \texttt{Mapped}\ f\ c} \qquad \frac{S \text{ maximal for } c}{S \text{ maximal for } \texttt{ContraMapped}\ f\ c}$$

$$\frac{S \text{ maximal for } c}{S \text{ maximal for } \texttt{Guarded}\ f\ c} \qquad \frac{S \text{ maximal for } c}{S \text{ maximal for } \texttt{Feedback}\ f\ c}$$

$$\frac{S \text{ maximal for } c_1 \qquad [c_1](S) \neq \emptyset}{S \text{ maximal for } \texttt{FirstOf}\ c_1\ c_2} \qquad \frac{S \text{ maximal for } c_1 \qquad S \text{ maximal for } c_2 \qquad [c_1](S) = \emptyset}{S \text{ maximal for } \texttt{FirstOf}\ c_1\ c_2}$$

$$\frac{S \text{ maximal for } c}{S \text{ maximal for } \texttt{Maximal}\ f\ c}$$

$$\frac{S \text{ maximal for } c \qquad \forall (c_i, g_i) \in [c](S)\ .\ S \text{ maximal for } c_i}{S \text{ maximal for } \texttt{Joined}\ c}$$

Proof. Each of the above derivation rules is trivially derived. □

Definition 4 (Open interaction stability)**.** Given a well-typed connector $c \in \mathcal{C}$ and system state $S \in \mathcal{S}$, an open interaction $\gamma = (x, g) \in \mathcal{O}$ is *stable in c and S* if and only if:

$$\forall S' \in \mathcal{S}. S \subseteq S' \Rightarrow \gamma \in [c](S')$$

Once enabled, a stable open interaction will always be proposed by the connector, regardless how many new port-atom pairs are activated.

Lemma 17. *Stability is a non-trivial property. There exists both stable and non-stable open interactions.*

Proof. Let our set of atoms be $\mathcal{A} = \{a_1, a_2\}$ and our set of ports be $P = \{p\}$. Consider the connector $c_1 = \texttt{Bind}\ a_1\ p$ and system state $S = \{((a_1, p), v)\} \in \mathcal{S}$, for some value $v \in \mathcal{V}$. Trivially, $\gamma = \{(v, \lambda d.\{((a, p), d)\})\} \in [c_1](S)$ is stable in c_1 and S. Thus, there exists stable open interactions.

Now, consider the connector $c_2 = \texttt{FirstOf}\ (\texttt{Bind}\ a_2\ p)\ (\texttt{Bind}\ a_1\ p)$. In the state $S = \{((a_1, p), v)\}$, we have that $\gamma = \{(v, \lambda d.\{((a_1, p), d)\})\} \in [c_2](S)$. Now consider $S' = \{((a_1, p), v), ((a_2, p), v)\}$. In the state S', where $S \subset S'$, the open interaction γ is no longer possible. Thus, there exists non-stable open interactions. This concludes the proof. □

Theorem 7. *For any atom $a \in \mathcal{A}$, port $p \in P$, connectors $c, c_1, c_2 \in \mathcal{C}$, system state $S \in \mathcal{S}$ and open interactions $\gamma, \gamma_1, \gamma_2 \in \mathcal{O}$, the following derivation rules hold:*

$$\frac{\gamma \in [\texttt{Bind}\ a\ p](S)}{\gamma \text{ stable in } \texttt{Bind}\ a\ p \text{ and } S} \qquad \frac{\gamma \in [\texttt{Success}\ v](S)}{\gamma \text{ stable in } \texttt{Success}\ v \text{ and } S}$$

$$\frac{\gamma \text{ stable in } c_1 \text{ and } S}{\gamma \text{ stable in } \texttt{OneOf}\ c_1\ c_2 \text{ and } S} \qquad \frac{\gamma \text{ stable in } c_2 \text{ and } S}{\gamma \text{ stable in } \texttt{OneOf}\ c_1\ c_2 \text{ and } S}$$

$$\frac{\gamma_1 = (x_1, g_1) \text{ stable in } c_1 \text{ and } S \qquad \gamma_2 = (x_2, g_2) \text{ stable in } c_2 \text{ and } S \qquad g_1 \text{ and } g_2 \text{ are downwards compatible}}{((x_1, x_2), \{x \mapsto g_1(x) \cup g_2(x) \mid x \in domain(g_1) \cap domain(g_2)\}) \text{ stable in } \texttt{BothOf}\ c_1\ c_2 \text{ and } S}$$

$$\frac{(x, g) = \gamma \text{ stable in } c \text{ and } S}{(f(x), g) \text{ stable in } \texttt{Mapped}\ f\ c \text{ and } S} \qquad \frac{(x, g) = \gamma \text{ stable in } c \text{ and } S}{(x, g \circ f) \text{ stable in } \texttt{ContraMapped}\ f\ c \text{ and } S}$$

$$\frac{(x, g) = \gamma \text{ stable in } c \text{ and } S \qquad f(x) = 1}{\gamma \text{ stable in } \texttt{Guarded}\ f\ c \text{ and } S}$$

$$\frac{(x, g) = \gamma \text{ stable in } c \text{ and } S}{(x, g \circ tag_x) \text{ stable in } \texttt{Feedback}\ c \text{ and } S}$$

$$\frac{\gamma \textit{ stable in } c_1 \textit{ and } S}{\gamma \textit{ stable in } \texttt{FirstOf}\ c_1\ c_2 \textit{ and } S}$$

$$\frac{\gamma \textit{ stable in } c_2 \textit{ and } S \qquad [c_1](S) = \emptyset \qquad S \textit{ maximal for } c_1}{\gamma \textit{ stable in } \texttt{FirstOf}\ c_1\ c_2 \textit{ and } S}$$

$$\frac{\gamma \textit{ stable in } c \textit{ and } S \qquad S \textit{ maximal for } c}{\gamma \textit{ stable in } \texttt{Maximal}\ f\ c \textit{ and } S}$$

$$\frac{\gamma \in [\texttt{Dynamic}\ p](S)}{\gamma \textit{ stable in } \texttt{Dynamic}\ p \textit{ and } S}$$

$$\frac{\begin{array}{c}\gamma_1 = (x_1, g_1) \textit{ stable in } c \textit{ and } S \qquad \gamma_2 = (x_2, g_2) \textit{ stable in } x_1 \textit{ and } S \\ g_1 \textit{ and } g_2 \textit{ are downwards compatible}\end{array}}{(x_2, \{x \mapsto g_1(x) \cup g_2(x) \mid x \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\}) \textit{ stable in } \texttt{Joined}\ c \textit{ and } S}$$

Proof. Each derivation rule of the above theorem is straightforwardly derived from the definitions of the semantics function $[\cdot]$, maximality and stability. □

Chapter 4

Implementation

In this chapter, we will cover our implementation of *BIP* in two functional programming languages, namely Haskell and Scala. However, in the first part of the chapter, we will present the general principles behind the two implementations. We will discuss in details the problematics related to the embedding of BIP as a DSL in a functional programming language. We will also present the general architectural approach that was taken. This approach, common to both frameworks, is very general and can be applied to a broad class of programming languages.

In the next two sections, we will focus on the Haskell and Scala implementations of *BIP*. Each language and each implementation is different enough so that having a separate discussion in each case feels appropriate.

For each language, we will describe the domain specific languages (DSLs) used to create atoms, connectors and complete *BIP* systems. Finally, we will describe the implementation of the engine running the *BIP* system.

The first, more general, part should be sufficient to get a good idea of how the frameworks are implemented. The two latter sections, which dive into the code, are a bit more involved but present the implementations in details.

4.1 General principles and architecture

In this section, we will expose the general principles behind the implementations we have made of BIP as an embedded domain specific language. The following exposition is not specific to any programming language.

4.1.1 Separation between user facing code and backend

In the implementations we have made of BIP, we make a strong distinction between the part of the framework that users can see and interact with, and the part of the framework responsible of the execution of systems. This separation between the user facing domain specific language and the engine is capital. This might sound trivial, but by having those two different parts independent, we are entirely free to change one part without affecting too much the other.

4.1.2 Domain Specific Language

We will begin our description of the implementation technique by focusing on the first kind of code, the domain specific language presented to the users of the frameworks.

The first important point to make is that, as we have established a strong separation between the domain specific language and the backend, the user facing DSL will not be responsible for the execution of anything. All actions performed within the DSL will simply either be recorded and later passed to the engine, or directly transmitted to the engine.

This will mean that, as can be clearly seen from the actual implementations, the user facing DSL code is trivial. It will merely consist of builder and façade classes[19] in language that support object-oriented design, such as Scala. In some other languages, such as Haskell, the user facing DSL will actually produce of a series of uninterpreted instructions, which should be interpreted within the engine. The actual details are not relevant at this point. The main point being that the DSL is simply a sophisticated façade for the actual engine.

Embedding level

Throughout the implementations, we will have to decide the level of embedding of the domain specific language, BIP, within the host languages. This decision will have many consequences regarding what can be done within and with the DSL. The two different levels of embedding are respectively called *shallow* and *deep*.

In a *shallow* embedding, expressions of the DSL correspond directly to expressions in the host language. This implementation technique of DSLs has the advantage of being very expressive and very easily implemented. However, expressions of the DSL being expressions of the language, they can generally not be inspected in any meaningful way. All the syntactic structure of an expression is lost when the host language

evaluates it. Therefore, the DSLs implementation has only access to evaluated expressions.

On the contrary, in a *deep* embedding, expressions and constructs of the DSL are translated to abstract syntax trees (ASTs), or similar syntactic structures, which can be further manipulated and then eventually evaluated. This technique is well-suited for analysis of the DSL programs, but may lack in expressivity and may be tedious to implement.

In our implementations, we have to balance between the two levels of embedding. Where possible, we take a deep embedding approach and encode expressions of the DSL as tree-like structures, which will then be able to manipulate, analyse and interpret. However, when the expressivity of the host language is needed, we will adopt the shallow embedding approach. For instance, we will always use Haskell and Scala functions where functions are needed. The alternative of creating our own structure to describe functions would be very tedious and would not interface well with the host language. This has the drawback that functions are completely opaque to us and can not be further analysed.

Atoms

A shallow embedding approach was taken for the behaviour of atoms, which should be as expressive as possible and appear as natural to the programmer as possible. This expressivity however has a cost, as the behaviour of atoms becomes completely opaque to the engine. For instance, we have no way of knowing what the atom will do in the future, whether they will await on a specific port or spawn new atoms. If such information were needed, a deeper embedding approach should be taken, which would result in a loss of expressivity. A compromise has to be made[1].

Connectors

In the frameworks, we take a deeper embedding approach for the connectors. Since we will want to analyse the structure of connectors and eventually manipulate them, it is very important to us that the underlying structure of connectors is preserved. To achieve this, we encode the connectors as tree-like structures, using algebraic data types. This structure will follow almost exactly the grammar of connector combinators that we have described.

Note that, in many cases, connectors will contain functions as members. For instance, the `Mapped` and `ContraMapped` combinator will contain functions that they will apply to respectively upwards and downwards values. Those functions will not be deeply embedded, and simply be functions of the host language. Once again, there is a tradeoff to be made between being able to analyse the code and being able to express it in a natural way.

[1]Promising techniques, such as *lightweight modular staging*[20], could potentially help reduce the syntactical overhead of having a deeper embedding. Those techniques have not been investigated as part of this thesis.

As this encoding of connectors as an algebraic data type is an internal implementation detail, it is important that we hide it by only providing the users of the framework with a library of derived combinators. This way, if we ever need to change the underlying structure of connectors, the user facing domain specific language doesn't need to change.

4.1.3 Engine architecture

So far we discussed principles behind the user-facing DSL of the frameworks. In this section, we will investigate the general architecture of the engine. The goal of the engine is to actually run the systems described by the DSL in a concurrent environment. In our approach, the engine is composed of several distinct components:

- A waiting list, in which the values sent by the atoms and their continuations are stored.
- A semantic function, which can return a stream of interactions from a connector and a waiting list.
- The main engine loop, which computes and executes interactions.
- Worker threads, which are each responsible for part of the execution of an atom.

Waiting list

The waiting list of the engine is a data structure that stores all the information about waiting atoms. The data structure has an entry for each atom-port pair where the atom is waiting on the port. The entry contains the following information:

- What has been sent by the atom through the port.
- What to do when a value is received. That is, the continuation of the atom.

This data structure supports the following four operations:

- Registering that an atom is waiting on a port. This simply adds an entry in the data structure.
- Registering that an atom is no longer waiting. This removes all entries associated to a given atom.
- Getting the entry associated to an atom and port.
- Getting all entries associated to a specific port.

As can be later seen in the implementations in Haskell and Scala of this data structure, it is usually supported by two mappings. The first mapping is composed of two layers, the first being indexed by atoms and the second by ports. This first mappings contains all the entries of the data structure. The second mapping, indexed by ports, contains only the identifier of atoms that are active for the port. Using those two mappings, the four operations we have described can be efficiently implemented.

Semantic function

As part of the backend is also found a semantic function, which allows the engine to obtain interactions from a connector and a waiting list. Before we can investigate the semantic function, we have to describe what an interaction is in the context of the frameworks.

Open interactions Just as in the formalisation, we make the distinction between open and closed interactions. An open interaction contains an upwards value and a downwards function. As a reminder, the upwards value is the value being propagated during the upwards phase. The downwards function, on the other hand, is responsible to propagate a value to all the involved atoms during the downwards phase. In the implementations, for convenience, we add a third field to open interactions. This extra field contains the set of all involved ports, which will be useful to know whether two interactions are downwards compatible or not.

Closed interactions The interactions returned to the engine by the semantic function are closed interaction. In this context, a closed interaction is simply a list of atoms with an associated task to perform. The task to be performed represents the continuation of the atom. Just as in the formalisation, closed interactions can be obtained from open interactions by feeding the upwards value to the downwards function.

Semantic function We can now look at how such interactions can be produced from connectors and waiting lists. As we have previously discussed, the connectors will be encoded as a tree-like structure. The semantic function will walk down this structure in a recursive way and combine open interactions. This procedure is performed in the exact same manner as was presented during the formalisation. The waiting list is only queried in the case of `Bind` and `Dynamic` combinators, to obtain information on waiting atoms. The final step of the semantic function simply consists of closing the open interactions to obtain a task for each involved atom.

Note that there is a major difference between the implementation and the formalisation of the semantics. In the implementations a stream of interactions is produced, where as in the formalisation a set of interactions is returned. As a reminder, a stream is a lazily computed ordered list, where elements at the head of the list can generally be returned without ever computing elements later in the list. This point is very relevant due to its implication regarding the performance of the semantic function. Indeed, it might not be feasible to return all interactions from a large connector, but returning only the first few of them can sometimes be performed very quickly. This has also more conceptual consequences that we will briefly discuss at the end of this thesis.

Main loop of the engine

So far, we have described the waiting list and semantic function of the engine. Those were in some sense the tools that are made at the disposition of the engine. We have

still to discuss how those tools are actually used. More specifically, we have to look at the active parts of the engine.

The first active part we investigate is the main loop of the engine. The goal of this loop is to repeatedly create the threads that must execute the behaviour of atoms and then trigger the next interaction when the system reaches a stable state. In more details, the loop performs over and over the same sequence of actions:

1. First of all, the main loop creates and starts all the threads necessary to execute the list of tasks to be performed. This list either comes from the initial configuration of the system in the first iteration of the loop, or from the interaction chosen at the end of the previous iteration.

2. Next, the engine waits for the system to be in a stable state, that is, when all threads have finished executing the behaviour of currently executing atoms. As we will see, the threads will terminate either when the atom has reached the end of its execution, or when an `await` instruction is performed. This waiting can be performed using synchronisation primitives offered by the language.

3. Once the system reaches a stable state, the main loop executes the semantic function and obtains a stream of interactions. If the stream is empty, the system immediately terminates, as no interactions are possible at this stage. This can either arise because all atoms have completely finished their execution, or because the system has entered a deadlock state.

4. If the execution continues, an interaction is then picked from the non-empty stream of interactions. By default, the first of those interaction is picked, as it is the one possible interaction the most efficiently computed. Other strategies can be chosen. In the actual frameworks, we give the option to the users to specify another function to select which interaction should be chosen.

5. Finally, all atoms part of the interaction are removed from the waiting list. The list of tasks from the interaction is specified to be executed during the next iteration of the loop, which immediately follows.

We will generally perform the main loop of the engine on the thread that started the system. Therefore, the procedure call that executes a system will block until the main loop terminates.

Worker Threads

Worker threads are the final piece of the engine. Each worker thread is momentarily responsible for the execution of the behaviour of an atom. They have the responsibly to answer calls to the various instructions that atoms can perform. The actions they must take will depend on the instruction being performed.

- When the atom requests that a new atom should be spawned, the worker thread intercept the call and spawns the child atom on a new worker thread. The identifier of the newly created atom is then communicated to the parent atom, which then continues its execution normally.

- When an `await` or `awaitAny` instruction is to be performed, the worker thread stores all the relevant information in the waiting list of the engine. For each port on which the atom is waiting, an entry is added to the waiting lists specifying which value was sent and what to do once a value is received. The worker thread then terminates its execution.

- When a request for the identifier of the running atom is issued, the worker thread simply provides the identifier of the atom and the execution continues normally.

At the end of its execution, the worker thread checks if the number of currently executing atoms has reached zero, which would mean that the system has reached a stable state. If it is the case, then the atom notifies the main loop, which can then continue its execution and compute the next interaction. It is very important that this check is performed even in the case of exceptions.

As they are frequently created and destroyed, the threads used by the frameworks should be as lightweight as possible. For performance reasons, lightweight threads, as opposed to heavy and generally OS-bound threads, should be used if the underlying platform supports them.

4.1.4 Final words

In this section, we have seen the general principles and architecture followed by the two implementations of BIP as an embedded domain specific language we have developed. The approach we have taken is largely language-agnostic and could be applied to other languages. In the next two following sections, we will see in details the two implementations we have made, starting with the Haskell implementation.

4.2 Haskell implementation

In the previous section, we have seen the general principles behind the implementation of BIP as an emdedded domain specific language. In this section, we will describe the implementation of *BIP* as a DSL in Haskell. For each concept, we first present the interface shown to the users of the framework and then discuss the internal implementation details. We take a top-down approach, starting from the definition of complete systems, then ports, atoms and finally connectors. Once this exposition has been done, we proceed to describe the actual engine behind the Haskell framework.

4.2.1 Systems

The main type of a BIP system is the `System` type, which is opaque to the users of the framework. Only three primitive instructions are available to describe systems.

- `newPort` which creates a new port. This instruction returns the unique identifier of the port. We will shortly discuss in more details what port identifiers are.

- `newAtom` which creates a new atom. This instruction, given the behaviour of the atom, returns its unique identifier. We will see later in this section what atom identifiers exactly are and how to define the behaviour of atoms.

- Finally, `registerConnector`, which indicates that a given connector should be used by the system. We will see in details later how connectors are implemented.

Monadic interface

Before we move on to ports and atoms, we have to discuss how to build systems using the three basic instructions we have just introduced. As is common place in Haskell, more complex systems are built by composition of simpler systems using the monadic interface. For instance, here is how a simple system of two ports and two atoms could be defined:

```
exampleSystem :: System s ()
exampleSystem = do
    p1 <- newPort
    p2 <- newPort
    a1 <- newAtom (someBehaviorUsing p1)
    a2 <- newAtom (someBehaviorUsing p2)
    registerConnector (someConnectorUsing a1 p1 a2 p2)
```

Where `someBehaviorUsing` and `someConnectorUsing` would be actions defined elsewhere. This very imperative looking syntax is just sugar for calls to functions from the `Monad` type class, from which `System` is an instance.

Monadic fix-point interface

Furthermore, systems also allow, through their `MonadFix` instance, users to refer to atoms before they are actually created. This is useful to be able to express systems of mutually recursive atoms. For instance, the following example system is able to refer to the atom `a2` before the instruction to create such an atom is performed.

```
exampleRecursiveSystem :: System s ()
exampleRecursiveSystem = mdo
    p1 <- newPort
    p2 <- newPort
    a1 <- newAtom (someBehaviorUsing a2 p1)
    a2 <- newAtom (someBehaviorUsing a1 p2)
    registerConnector (someConnectorUsing a1 p1 a2 p2)
```

This feat is made possible by the lazy evaluation strategy of Haskell. Using the `mdo` keyword, the thunks that will hold actual identifiers created by the system are maid available in the entire system expression. This can be thought of as a generalisation of fix points to monads.

Implementation of `System`

So far, we have presented the interface of systems and instructions to build systems. The actual implementation details behind are completely invisible to users. The actual `System` type is defined as follows:

```
-- | Describes a BIP system.
--
--   The type 's' ensures that identifiers of a system
--   may not leave the scope of the system.
newtype System s a = System
    { getSystem :: State (InitialSystemState s) a }
    deriving (Functor, Applicative, Monad, MonadFix)

-- | Contains the initial state of a system, that is
--   the state of a system before it is actually started.
data InitialSystemState s = InitialSystemState
    { getConnector :: ClosedConnector s
    -- ^ Indicates which connector is to be used by the system.
    , getTasks     :: [Task s]
    -- ^ Indicates, for each atom which action must be executed.
    , getNextId    :: Integer
    -- ^ Indicates the next free identifier.
    }

-- | Task to be executed by an atom.
```

```
data Task s = Task
    { getAtom   :: AtomId s
    -- ^ The atom responsible to execute the action.
    , getAction :: Action s ()
    -- ^ The action to execute.
    }

-- | Contains a single connector whose upwards and downwards type,
--   whatever they may be, are the same.
data ClosedConnector s where
    ClosedConnector :: Connector s a a -> ClosedConnector s
```

The system type is defined as a `State` monad, whose state is called `InitialSystemState`. Expressed differently, this means that all a `System` can do is build under the hood a value of type `InitialSystemState` which will contain all information needed to actually start executing a system. As can be seen above, an `InitialSystemState` contains:

- A connector, whose upwards and downwards types are the same.
- Actions to be executed by each atoms.
- The next free integer to use as an identifier.

Remark. As explained in the comments, the parameter type s of `System` and many other types that we have seen or that we will introduce later, is there to ensure that identifiers do not escape the scope of the system.

As we will later see, the only function that can actually execute systems will require the type parameter `s` of `System` to be universally quantified. This ensures that identifiers can not leave the scope of the system they were created in without producing a type error at compile time. This exact same trick was used to implement the `ST` monad[21], which must also ensure that identifiers do not leave some restricted scope to preserve type safety.

The reason we do not want identifiers to escape the scope of their system is very simple. It is due to the fact that different systems may create ports identifiers with the same underlying number but different upwards and downwards types. To motivate the problem, imagine that two ports with the same underlying identification number but different types were to be used in the same system. A value to be received by the first port could then instead be received by the second (since they are equal to the engine). The second port would then receive a value of a different type than expected. This would break type safety.

Implementation of the three `System` instructions

Using the above representation of `System`, here is how the three basic instructions to build systems are defined:

```
-- |Creates a new atom.
newAtom :: Action s a -> System s (AtomId s)
newAtom x = System $ do
    s <- get
    let n  = getNextId s    -- Next Id to use.
        as = getActions s   -- Actions to execute so far.
        i  = AtomId n       -- Id of the newly created atom.
        s' = s
            { getNextId = succ n   -- Specifying the next Id to use.
            , getActions = Task i (void x) : as   -- New task.
            }
    put s'
    return i

-- | Creates a new port.
newPort :: System s (PortId s d u)
newPort = System $ do
    s <- get
    let n = getNextId s
    put $ s { getNextId = succ n }
    return $ PortId n

-- | Registers a connector for the system.
--
--   Any previous call to 'registerConnector' will get overwritten.
registerConnector :: Connector s a a -> System s ()
registerConnector c = System $ do
    s <- get
    put $ s { getConnector = ClosedConnector c }
```

4.2.2 Ports

In the Haskell framework, ports are simply identifiers that also contain information about the type of values that can be sent and received through the port.

```
-- | Port identifier.
--
--   * 's' is a phantom type ensuring that identifiers
--     do not escape the scope of the system.
--
--   * 'd', for downwards, is the type of values that
--     can be received by the port.
--
--   * 'u', for upwards, is the type of values that
```

```
--     can be sent through the port.
newtype PortId s d u = PortId Integer
    deriving (Eq, Ord, Show)
```

Remark. The three type parameters of `PortId` are *phantom* types, that is, types in the signature that are not used in the right hand side of the definition. Those types are there to ensure that the identifiers can only be used in permitted contexts.

At the user level, the only way to create a new port identifier is be through the `newPort` instruction of systems. This will ensure that all port identifiers are unique in the given system. This also has the effect that port identifiers created through this mean have their type fully instantiated, which ensure that they are *monomorphic*[2]. This makes sure that users have no way to break type safety by using an identifier created in a different system or by using an identifier in two different and contradicting contexts.

4.2.3 Atoms

After ports, we have to investigate atoms, which are conceptually the active processes of BIP systems.

Atom identifiers

Just as ports, atoms are uniquely identifiable. Atom identifiers are defined as:

```
-- | Atom identifier.
newtype AtomId s = AtomId Integer
    deriving (Eq, Ord, Show)
```

Just as with ports, users of the framework will not have access to the `AtomId` constructor. Their only way to obtain an atom identifier is through the use of the `newAtom` instruction.

Atom actions

The behaviour of an atom consists of a (possibly infinite) series of instructions to execute. Each instruction can either:

- Perform some arbitrary computation and input/output in the `IO` monad, using the `liftIO` instruction. The instruction returns the value computed by the `IO` action.
- Wait on ports, using either the `await` or `awaitAny` instructions. Those instructions return the value received by the port (or one of the ports in the case of `awaitAny`) once an interaction involving the atom takes place.

[2] A monomorphic value, contrary to a polymorphic one, only has a single type.

- Spawn new atoms, using the `spawn` instruction. This instruction returns the identifier of the newly created atom.
- Get the identifier of the atom itself, using the `getSelfId` instruction.

As before with systems, those actions can be combined using the monadic interface. For instance, here is the behaviour of an atom that reads a line `x` from standard input, sends the value `x` to a port `p` and prints the value `v` received through the port to standard output:

```
exampleAction :: Action s ()
exampleAction = do
    x <- liftIO getLine
    v <- await p x
    liftIO $ print v
```

The users of the framework are only exposed to the interface we have just presented. After having discussed this user-facing interface, we will now investigate how `Action` is implemented. Here is how the actual `Action` type is implemented:

```
-- | Action performed by atoms.
newtype Action s a = Action
    { getInstructions :: ProgramT (Instruction s) IO a }
    deriving (Functor, Applicative, Monad, MonadIO)

-- | Possible instructions to be performed within actions.
data Instruction s a where
    Spawn :: Action s a -> Instruction s (AtomId s)
    Await :: [AwaitCase s a] -> Instruction s a
    GetId :: Instruction s (AtomId s)

-- | Contains information about the case of an 'Await' instruction.
--
--   Each record contains the identifier of the port on which to wait,
--   the value to be sent, and a function to apply on the downwards function.
data AwaitCase s a where
    AwaitCase :: PortId s d u -> u -> (d -> a) -> AwaitCase s a

instance Functor (AwaitCase s) where
    fmap f (AwaitCase p x g) = AwaitCase p x (f . g)

-- | Sends a value on the given port and waits to receive a value.
await :: PortId s d u -> u -> Action s d
await p x = Action $ singleton $ Await [AwaitCase p x id]

-- | Sends values on the given ports and
```

```
--   waits to receive a value on any of the ports.
awaitAny :: [AwaitCase s a] -> Action s a
awaitAny as = Action $ singleton $ Await as

-- | Specifies what to send on the particular port.
onPort :: PortId s d u -> u -> AwaitCase s d
onPort p x = AwaitCase p x id

-- | Spawns a new atom.
spawn :: Action s d -> Action s (AtomId s)
spawn a = Action $ singleton $ Spawn a

-- | Returns the identifier of the atom.
getSelfId :: Action s (AtomId s)
getSelfId = Action $ singleton $ GetId
```

An action is simply defined as a series of instructions, each instruction being either spawning a new atom, awaiting on ports or getting the identifier of the atom. The possibility to perform `IO` action is directly built into the underlying `ProgramT (Instruction s) IO` type. This implementation of `Action` makes heavy use of the `operational` package[22], which defines the `ProgramT` monad transformer.

As we have just discussed, the `Action` type merely contains a series of instructions. Those instructions will be have to be interpreted for any actual actions to be executed. We will discuss this in detail later in this section when we discuss the implementation of the engine.

4.2.4 Connectors

The final components of BIP systems we have yet to describe are connectors. The implementation of connectors in Haskell will follow directly the formalisation we have given of them in the dedicated chapter.

Algebraic data type for connectors

The connector type is declared as a generalised algebraic data type as follows:

```
-- | Connects together the different ports of a system.
--
--   * 's' corresponds to the type phantom type of the system,
--     which ensures that identifiers do not escape the system
--     in which they are defined.
--
--   * 'd' is the type of values propagated downwards by the connector.
--
--   * 'u' is the type of values progagated upwards by the connector.
```

```
data Connector s d u where
    -- Core combinators
    Bind    :: AtomId s -> PortId s d u -> Connector s d u
    Success :: u -> Connector s d u
    Failure :: Connector s d u
    OneOf   :: Connector s d u -> Connector s d u -> Connector s d u
    BothOf  :: Connector s d u -> Connector s d v -> Connector s d (u, v)

    -- Data combinators
    Mapped       :: (u -> v) -> Connector s d u -> Connector s d v
    ContraMapped :: (e -> d) -> Connector s d u -> Connector s e u
    Guarded      :: (u -> Bool) -> Connector s d u -> Connector s d u
    Feedback     :: Connector s (d, u) u -> Connector s d u

    -- Priority combinators
    Maximal :: (u -> u -> Ordering) -> Connector s d u -> Connector s d u
    FirstOf :: Connector s d u -> Connector s d u -> Connector s d u

    -- Dynamic combinators
    Dynamic :: PortId s d u -> Connector s d u
    Joined  :: Connector s d (Connector s d u) -> Connector s d u
```

This representation allows us to actually manipulate connectors as a tree-like structure. Being algebraic data types, connectors can be deconstructed via pattern matching. As we will see, this will be performed within the engine to interpret the semantic function on a connector or to give connectors an optimised structure.

In addition, encoding the connectors as a generalised algebraic data type gives us very strong static guarantees through the help of the type system. In particular, the types will ensure that all operations performed by the different connectors are valid. Note that all constructors have been given a type that corresponds, in the formalisation of connectors, to the type given to the corresponding connector combinator.

Type class instances

Connectors are instances of many interesting Haskell type classes. The following instance are justified by the algebraic properties of the connector combinators that we have proven in the dedicated chapter.

```
instance Functor (Connector s d) where
    fmap f c = Mapped f c

instance Profunctor (Connector s) where
    lmap f c = ContraMapped f c
    rmap f c = Mapped f c
```

```
instance Applicative (Connector s d) where
    pure x = Success x
    c1 <*> c2 = Mapped (uncurry ($)) $ BothOf c1 c2

instance Alternative (Connector s d) where
    empty = Failure
    c1 <|> c2 = OneOf c1 c2

instance Monad (Connector s d) where
    return x = Success x
    c >>= f = Joined (Mapped f c)
    c1 >> c2 = Mapped snd $ BothOf c1 c2
    fail _ = Failure

instance MonadPlus (Connector s d) where
    mzero = Failure
    mplus c1 c2 = OneOf c1 c2

instance Monoid u => Monoid (Connector s d u) where
    mempty = Success mempty
    mappend c1 c2 = Mapped (uncurry mappend) $ BothOf c1 c2
```

The type class instances correspond to the various algebraic concepts in the following way:

- The `Functor` instance is motivated by the covariant functor F_{up}.
- The `Profunctor`[23] instance corresponds to the combination of the contravariant functor F_{down} and the covariant functor F_{up}.
- The `Applicative`[24] instance is motivated by the $(F_{up}, \eta^{\mathcal{V}}, \mu^{\mathcal{V}})$ monad, `Applicative` being a (logical) subclass of `Monad`.
- The `Alternative` instance corresponds to the $(\mathcal{C}, \texttt{OneOf}, \texttt{Failure})$ monoid.
- The `Monad`[25] instance corresponds to the $(F_{up}, \eta^{\mathcal{V}}, \mu^{\mathcal{V}})$ monad.
- The `MonadPlus` instance, being equivalent to `Alternative`, is also justified by the $(\mathcal{C}, \texttt{OneOf}, \texttt{Failure})$ monoid.
- Finally, the `Monoid` instance corresponds to the $(\mathcal{C}, \texttt{Mapped} \times (\texttt{BothOf} \cdot \cdot), \texttt{Success}\ 1)$ monoids, where $\times$ and 1 are respectively the binary operation and neutral element of some monoid.

Monadic do-notation for connectors

As we have just seen, connectors also have a `Monad` instance. This means that the monadic `do`-notation can be used to describe connectors. For instance, below is shown a connector that synchronises three connectors and provides upwards the sum of the three upwards value:

```
connector = do
    v1 <- connector1
    v2 <- connector2
    v3 <- connector3
    return (v1 + v2 + v3)
```

This particular connector is translated into the following *desugarised* version.

```
connector =
    Joined (Mapped (\ v1 ->
        Joined (Mapped (\ v2 ->
            Joined (Mapped (\ v3 ->
                Success (v1 + v2 + v3)
            ) connector3)
        ) connector2)
    ) connector1)
```

The former version is arguably simpler than the latter! Moreover, this syntactic sugar comes for free once the instance of `Monad` has been defined. We will make use of this particular syntactic nicety in the some of the example applications.

Connector library

The constructors of the `Connector` type defined above are not made available to the users of the framework. Instead, users have to use a library of combinators derived from those primitive constructors. This ensures that the underlying representation of connectors can be changed without changing the interface presented to the users. In addition to the many functions derived from the type class instances of connectors, the following combinators are provided to the users. This also gives us the opportunity to rename some of the combinators to give them friendlier, domain specific, names for the user of the library.

```
-- | Connects the given port to the specified atom.
bind :: AtomId s -> PortId s d u -> Connector s d u
bind a p = Bind a p

-- | Chooses one of the given connector non-deterministically.
anyOf :: [Connector s d u] -> Connector s d u
anyOf cs = foldr OneOf Failure cs
```

```
-- | Synchronizes all underlying connectors and collects all upwards values.
allOf :: [Connector s d u] -> Connector s d [u]
allOf cs = mconcat $ fmap (fmap (: [])) cs

-- | Involves any number of the underlying connectors.
manyOf :: [Connector s d u] -> Connector s d [u]
manyOf cs = mconcat [fmap (: []) c <|> pure [] | c <- cs]

-- | Behaves as the first enabled connector of the given list.
firstOf :: [Connector s d u] -> Connector s d u
firstOf cs = foldr FirstOf Failure cs

-- | Applies a function of upwards values.
upwards :: (u -> v) -> Connector s d u -> Connector s d v
upwards f c = Mapped f c

-- | Always uses the specified upwards value.
sending :: v -> Connector s d u -> Connector s d v
sending x c = upwards (const x) c

-- | Applies a function of downwards values.
downwards :: (e -> d) -> Connector s d u -> Connector s e u
downwards f c = ContraMapped f c

-- | Always uses the specified downwards value.
receiving :: d -> Connector s d u -> Connector s e u
receiving x c = downwards (const x) c

-- | Ensures that a given predicate holds on upwards values.
ensuring :: (u -> Bool) -> Connector s d u -> Connector s d u
ensuring f c = Guarded f c

-- | Feeds the upwards value downwards.
feedback :: Connector s (d, u) u -> Connector s d u
feedback c = Feedback c

-- | Closes the connector, using the upwards value as downwards value.
close :: Connector s a a -> Connector s d ()
close c = sending () $ feedback $ downwards snd c

-- | Ensures that the upwards value is maximal amongst the possible
--   upwards values.
maximal :: Ord u => Connector s d u -> Connector s d u
```

```haskell
maximal c = Maximal compare c

-- | Ensures that the upwards value is minimal amongst the possible
--   upwards values.
minimal :: Ord u => Connector s d u -> Connector s d u
minimal c = Maximal (comparing Down) c

-- | Ensures that the upwards value is maximal amongst the possible
--   upwards values.
maximalBy :: (u -> u -> Ordering) -> Connector s d u -> Connector s d u
maximalBy f c = Maximal f c

-- | Ensures that the upwards value is minimal amongst the possible
--   upwards values.
minimalBy :: (u -> u -> Ordering) -> Connector s d u -> Connector s d u
minimalBy f c = Maximal (\ a b -> inverse $ f a b) c
  where
    inverse LT = GT
    inverse EQ = EQ
    inverse GT = LT

-- | Connects the given port non-deterministically to any atom.
dynamic :: PortId s d u -> Connector s d u
dynamic p = Dynamic p
```

So far, we have seen in some details the interface provided by the framework to its users. We have also seen that system descriptions written by users can be evaluated to the obtain the initial configuration of the system. We have then discussed how the behaviour of atoms can be seen as series of unevaluated primitive instructions. Finally, we have also seen that connectors are encoded as a tree-like structure, which can be inspected and interpreted. We are in a situation where we have in some sense described the *grammar* of the framework. However, we have yet to discuss how to interpret those representations to actually execute the represented system. In other words, we have yet to discuss the *semantics*.

In the rest of this section, we will introduce the actual engine that gives BIP systems their semantic meaning in terms of an actual running concurrent system. Given the representation of a system as we have just discussed, the engine will be able to actually execute the system in a concurrent way.

4.2.5 Interactions

Just as when we have formalised the semantics of combinators, we begin by describing the concept of *interactions* and *open interactions*.

Closed interactions

In the context of this framework, an interaction is simply a list of tasks (`Task`), each of which consisting of an atom and an action to be executed. This action represents the rest of the computation of the atom.

```
-- | Interaction.
type Interaction s = [Task s]
```

Open interactions

Open interactions, just as in the formalisation, consist of an upwards value and downwards function. For convenience, and to avoid having to inspect functions, the set of involved atoms is also explicitly present as a field.

```
-- | Open interaction.
data OpenInteraction s d u = OpenInteraction
    { getUpwards   :: u
    -- ^ Value propagated upwards by the interaction.
    , getDownwards :: d -> [Task s]
    -- ^ Given a downwards value, returns all continuations to perform.
    , getInvolved  :: Set (AtomId s)
    -- ^ The set of all atoms involved in the interaction.
    }
```

As seen during the formalisation, it is possible to obtain a (closed) interaction from an open interaction using the `close` function.

```
-- | Obtains an interaction from an open interaction.
--
--   This function feeds the upwards value of the open interaction
--   to its downwards function.
close :: OpenInteraction s a a -> Interaction s
close (OpenInteraction x f _) = f x
```

4.2.6 Waiting list

We must also somehow represent the state of waiting atoms. For the purpose, we have implemented a waiting list of atoms. This data structure will capture, for every atom waiting on some ports, what has been sent through the port and what to do once a value is received. The data structure supports the following operations:

- Registering an atom as waiting on a port.
- Registering an atom as running, that is, no longer waiting on any port.
- Getting the open interaction, if any, corresponding to a given atom-port pair.
- Getting all open interactions involving a given port.

Data structure definition

Before we actually show the implementation of the different operations, let us discuss how the data structure is implemented.

```
-- For clarity, we differentiate between integers corresponding to
-- AtomIds and to PortIds.
type AtomInteger = Integer
type PortInteger = Integer

-- | Data structure containing information about waiting atoms.
data WaitingList s = WaitingList
    { getFromAtoms :: Map AtomInteger (Map PortInteger (WaitState s))
    -- ^ For each atom, indicates which ports are active.
    --   For each active port, it also indicates what value was sent
    --   and what to do with the received value.
    , getFromPorts :: Map PortInteger (Set AtomInteger)
    -- ^ For each port, indicates which atoms are waiting on the port.
    }

-- | Contains an upwards value of some unknown type and
--   a downwards function whose domain is an unknown type.
data WaitState s where
    WaitState :: u -> (d -> Action s ()) -> WaitState s
```

A waiting list consists of two mappings, which together allow the engine to efficiently perform the four previously specified instructions.

Remark. You may have noticed that the `WaitState` constructor completely forgets about the upwards and downwards types `u` and `d`. This is necessary here because the `Map` container is not heterogeneous, which means that a mapping can only contain values of the same type. However, as we will see, the type information about `u` and `d` are also contained in the `PortId` type, which will allow the engine to coerce[3] the upwards and downwards value to the correct type. It is for this reason that we statically ensured that all `PortId` are monomorphic and that ports of one system could not be used in any other system.

Operations

Now that we have introduced the data structure, we can discuss how the four operations on it are defined.

[3]Coercion can be done in Haskell using the function `unsafeCoerce`. This function, whose type is `a -> b` can coerce a value from any type to any other type. Obviously dangerous, this function, when used incorrectly, may break all type safety properties. In the framework however, we ensured that the particular use we make of the function is safe.

```haskell
-- | Registers in the waiting list that an atom is waiting on a given port.
setWaiting :: AtomId s -> PortId s d u -> u -> (d -> Action s ())
           -> WaitingList s -> WaitingList s
setWaiting (AtomId a) (PortId p) u d (WaitingList as ps) = WaitingList as' ps'
  where
    as' = Map.insertWith Map.union a (Map.singleton p $ WaitState u d) as
    ps' = Map.insertWith Set.union p (Set.singleton a) ps

-- | Registers in the waiting list that an atom is no longer waiting.
setRunning :: AtomId s -> WaitingList s -> WaitingList s
setRunning (AtomId a) (WaitingList as ps) = WaitingList as' ps'
  where
    -- Getting all ports active for the given atom.
    activePorts = Map.keysSet $ Map.findWithDefault Map.empty a as
    -- Remove the entry for the atom.
    as' = Map.delete a as
    -- Updates the entries of all ports previously associated to the atom.
    ps' = foldl' removeAtom ps activePorts
      where
        -- Indicates that the atom is no longer waiting on the
        -- port p, by removing it from the set of active atoms
        -- for the given port.
        -- If the port is no longer active for any atom,
        -- its entry in the map is removed.
        removeAtom m p = Map.update shrink p m
        shrink s = let s' = Set.delete a s in
            if Set.null s' then Nothing else Just s'

-- | Returns the interaction, if any, related to a specific atom and port pair.
getBoundInteraction :: AtomId s -> PortId s d u -> WaitingList s
                    -> Maybe (OpenInteraction s d u)
getBoundInteraction (AtomId a) (PortId p) (WaitingList as _) = do
    ps <- Map.lookup a as
    WaitState u d <- Map.lookup p ps
    -- Coercing upwards value and downwards functions to the correct type.
    -- This is necessary as WaitState loses all type information.
    -- Fortunately, those types are recovered from the type of the port.
    let tu = unsafeCoerce u :: u
        td = unsafeCoerce d :: (d -> Action s ())
    return $ OpenInteraction
        { getUpwards   = tu
        , getDownwards = \ x -> [Task (AtomId a) (td x)]
        , getInvolved  = Set.singleton (AtomId a) }
```

```haskell
-- | Returns all interactions related to the given port.
getBoundInteractions :: PortId s d u -> WaitingList s
                     -> [OpenInteraction s d u]
getBoundInteractions (PortId p) w = do
    a <- Set.toList $ Map.findWithDefault Set.empty p $ getFromPorts w
    return $ fromMaybe e $ getBoundInteraction (AtomId a) (PortId p) w
  where
    e = error "getBoundInteractions: No interaction for an active atom-port."
```

4.2.7 Semantic function

Using the waiting list data structure and operations we have just defined, we can finally implement the semantics of the connectors. The semantic function, `getInteractions`, corresponds to the $[\cdot]$ function of the formalisation.

```haskell
-- | Gets the list of possible interactions. The list is computed lazily,
--   element by element.
getInteractions :: Connector s d u -> WaitingList s -> [OpenInteraction s d u]
getInteractions c w = case c of
    -- Core combinators
    Success x    -> [(OpenInteraction x (const []) (Set.empty))]
    Failure      -> []
    Bind a p     -> maybeToList $ getBoundInteraction a p w
    OneOf c1 c2  -> getInteractions c1 w ++ getInteractions c2 w
    BothOf c1 c2 -> do
        o1 <- getInteractions c1 w
        o2 <- getInteractions c2 w
        let o3 = OpenInteraction
                { getUpwards   = (getUpwards o1, getUpwards o2)
                , getDownwards = \ x -> getDownwards o1 x ++ getDownwards o2 x
                , getInvolved  = Set.union (getInvolved o1) (getInvolved o2)
                }
        -- Ensures that the sets of involved atoms are disjoint.
        guard (Set.size (getInvolved o3) ==
               Set.size (getInvolved o1) +
               Set.size (getInvolved o2))
        return o3

    -- Data combinators
    Mapped f c1 -> do
        o1 <- getInteractions c1 w
        return $ o1 { getUpwards = f (getUpwards o1) }
    ContraMapped f c1 -> do
        o1 <- getInteractions c1 w
```

```
        return $ o1 { getDownwards = getDownwards o1 . f }
    Guarded f c1 -> do
        o1 <- getInteractions c1 w
        guard (f $ getUpwards o1)
        return o1
    Feedback c1 -> do
        o1 <- getInteractions c1 w
        return $ o1 { getDownwards =
            \ x -> getDownwards o1 (x, getUpwards o1) }

    -- Priority combinators
    FirstOf c1 c2 -> case getInteractions c1 w of
        []  -> getInteractions c2 w
        os1 -> os1
    Maximal f c1 ->
        let os1 = getInteractions c1 w
            upwardsValues = fmap getUpwards os1
            biggerThan x = \ y -> f y x == GT
            isMaximal x = not $ any (biggerThan x) upwardsValues
        in filter (isMaximal . getUpwards) os1

    -- Dynamic combinators
    Dynamic p -> getBoundInteractions p w
    Joined c1 -> do
        o1 <- getInteractions c1 w
        o2 <- getInteractions (getUpwards o1) w
        let o3 = OpenInteraction
                { getUpwards   = getUpwards o2
                , getDownwards = \ x -> getDownwards o1 x ++ getDownwards o2 x
                , getInvolved  = Set.union (getInvolved o1) (getInvolved o2)
                }
        -- Ensures that the sets of involved atoms are disjoint.
        guard (Set.size (getInvolved o3) ==
               Set.size (getInvolved o1) +
               Set.size (getInvolved o2))
        return o3
```

This function will be used by the engine when the system is in a stable stable, that is when not a single atom is running, to obtain the next possible interactions. We will see in the chapter on optimisations how to redefine this function to avoid some performance issues. We will also investigate how to get stable interactions[4], which will allow the engine to execute interactions even though there are atoms still running,

[4]Remember that stable interactions are, given a system state, interactions that will still be offered regardless how many new ports are activated.

while ensuring that the priority policy is respected.

Also note that the function computes the list of interactions in a *lazy* way, meaning that each elements of the list can potentially be computed without looking at the elements later in the list. This is a very important performance point, since we can for instance get the first possible interaction without worrying about any other interactions.

Closed semantics

From the open semantics we can define, as we have done during the formalisation, the closed semantics. This function is defined as follows in the framework:

```
getClosedInteractions :: ClosedConnector s -> WaitingList s -> [Interaction s]
getClosedInteractions (ClosedConnector c) w = fmap close $ getInteractions c w
```

4.2.8 Engine

We finally reach the point where we can actually discuss the implementation of the engine itself. The engine will manage the different actions to execute, manage threads and execute interactions. During the runtime of the system, the engine will need to maintain three important variables:

- The number to be used as identifier by the next atom to be spawned.
- The number of atoms currently executing.
- The waiting list of the system, which contains information about all atoms that are waiting on some of their ports.

In addition, the system will also have at is disposition a semaphore[3], which will be used to block the main thread of execution while there are still atoms running. For convenience, all those variables are stored in a single record of the following type:

```
-- | Contains mutable information about the state of the running system.
data SystemState s = SystemState
    { getNextIdVar       :: MVar Integer
    -- ^ Variable holding the integer to use as next identifier.
    , getRunningVar      :: MVar Integer
    -- ^ Variable counting the number of currently executing atoms.
    , getWaitingListVar :: MVar (WaitingList s)
    -- ^ Variable containing the waiting list.
    , getSemaphore       :: MVar ()
    -- ^ Semaphore used to block the main thread while the system is running.
    }
```

Note on MVars

As you may have noticed, the above definition of the state of the systems uses mutable variables in the form of MVar[26]. An MVar, pronounced *em-var*, represents a mutable variable which can either be full or empty. MVars support the following operations:

- Taking the value out of a full variable, takeMVar. The operation blocks if the variable is empty.
- Putting a value in an empty variable, putMVar. The operation blocks if the variable is already full.
- Putting a value in a variable, tryPutMVar. The operation doesn't block, but is without effect if the variable is already full.

MVars have been especially designed to work well in a concurrent environment. In particular, they have the following properties:

- Operations, such as taking the value of an MVar or putting a value in an MVar are *atomic*, meaning that their effect appears to take place at a single point in time. Moreover, the effects of a sequence of such operations will always be seen as taking place in order.
- Trying to acquire the value of an empty MVar or trying to put a value in an already full MVar will block until the operation is possible. A fairness property guarantees that eventually no operations will block indefinitely, given that the MVar is not hold indefinitely.

Other instructions on MVars

Using the above primitive operations on MVars, we build the following functions, which we will use within the engine.

```
-- | Increments the value inside of an 'MVar'.
--   Returns the value present before the increment takes place.
--
--   Will block on empty 'MVar's.
fetchAndIncrement :: MVar Integer -> IO Integer
fetchAndIncrement v = do
    n <- takeMVar v
    let !n' = succ n
    putMVar v n'
    return n

-- | Increments the value inside of an 'MVar'.
--
--   Will block on empty 'MVar's.
```

```
increment :: MVar Integer -> IO ()
increment v = incrementBy v 1

-- | Increments the value inside of an 'MVar' by a certain amount.
--
--   Will block on empty 'MVar's.
incrementBy :: MVar Integer -> Integer -> IO ()
incrementBy v k = do
    n <- takeMVar v
    let !n' = n + k
    putMVar v n'

-- | Decrements the value inside of an 'MVar'.
--   Returns the value after the decrement takes place.
--
--   Will block on empty 'MVar's.
decrementAndFetch :: MVar Integer -> IO Integer
decrementAndFetch v = do
    n <- takeMVar v
    let !n' = pred n
    putMVar v n'
    return n'

-- | Waits until the 'MVar' is full.
--
--   Will block on empty 'MVar's.
wait :: MVar () -> IO ()
wait v = takeMVar v

-- | Ensures that an 'MVar' is full.
--
--   This will not block if the variable is already full.
signal :: MVar () -> IO ()
signal v = void $ tryPutMVar v ()
```

The `runSystem` function

We now have all the pieces of the puzzle and can finally introduce the `runSystem` function, which given a system description, executes the BIP system. Let us first look at its type:

```
runSystem :: (forall s. System s ()) -> IO ()
```

As hinted before, this type is somewhat particular. The type of `runSystem` ensures that the type parameter `s` of its first argument is universally quantified. As previously

explained, this ensures that identifiers of a system can not be used in another system, since then both would need to share the same type parameter `s` and therefore wouldn't be universally quantified anymore. The function is defined as follows.

```
-- | Executes the given BIP system.
runSystem :: (forall s. System s ()) -> IO ()
runSystem s = runSystemWith defaultOptions s
```

As can be seen, the `runSystem` function merely calls the `runSystemWith` function, with a default set of options. For now, the options are defined as follows:

```
-- | Options of the system.
data SystemOptions = SystemOptions
    { getInteractionPicker :: forall s. [Interaction s] -> IO (Interaction s)
    -- ^ Picks an interaction from a non-empty list of interactions.
    }

-- | Default set of options.
defaultOptions :: SystemOptions
defaultOptions = SystemOptions
    { getInteractionPicker = return . head }
```

The default set of options arbitrarily specifies that the first interaction should be chosen. As alternatives, users of the framework may choose to execute an interaction which leads to the *maximal progress*[5] or rely on the user to pick an interaction in an interactive manner, or provide any of their own functions.

We will now look at the implementation of the `runSystemWith` function. Since is it longer than usual, it is logically separated in multiple functions. The function is defined as follows:

```
-- | Excutes the given BIP system using custom options.
runSystemWith :: SystemOptions -> (forall s. System s ()) -> IO ()
runSystemWith opts s = do
    -- Obtaining the connector, the actions to execute, as well
    -- as the next id to use, from the system description.
    let i  = execState (getSystem s)
                (InitialSystemState (ClosedConnector Failure) [] 0)
        c  = getConnector i
        ts = getTasks i
        n  = getNextId i

    -- Initialising the state of the system.
    nextIdVar      <- newMVar n
```

[5]An interaction which leads to maximal progress is defined as an interaction that involves the most number of atoms from the possible interactions.

```
    runningVar     <- newMVar 0
    waitingListVar <- newMVar emptyWaitingList
    semaphore      <- newEmptyMVar

    let state = SystemState
            { getNextIdVar       = nextIdVar
            , getRunningVar      = runningVar
            , getWaitingListVar = waitingListVar
            , getSemaphore       = semaphore }

    -- Executing the engine, starting with the given tasks.
    when (not $ null ts) $ engineLoop opts c state ts
```

The function simply evaluates the system description to obtain the connector which should be used, the tasks to initially execute and the next free identifier. At the end of the function, a call to the main engine loop is made. If there are no tasks, meaning that the system doesn't have any atom, the main loop isn't called and the function terminates immediately.

Engine loop

The main loop of the engine is defined as follows.

```
-- | Main loop of the engine.
engineLoop :: SystemOptions -> ClosedConnector s -> SystemState s
           -> [Task s] -> IO ()
engineLoop opts c state as = do
    -- Sets the correct number of running atoms.
    incrementBy (getRunningVar state) (toInteger (length as))

    -- Spawning all atoms.
    forM_ as (spawnAtom state)

    -- Waiting on the semaphore to be notified.
    -- This means that we can now try to execute the next interaction.
    wait (getSemaphore state)

    -- Reading the current state of the waiting list.
    w <- takeMVar (getWaitingListVar state)

    -- Trying to get an interactions
    case filter (not . null) $ getClosedInteractions c w of
        [] -> return ()  -- No non-empty interactions to perform.
                         -- Might be a deadlock, or not, depending on
                         -- the state of the waiting list.
```

```
        is -> do
            -- We have possibly multiple interactions.
            -- We let the interaction picker decide.
            i <- getInteractionPicker opts is

            -- Update the waiting list.
            -- All atoms of the interactions are no longer waiting.
            putMVar (getWaitingListVar state) $
                foldl' (flip setRunning) w $ fmap getAtom i

            -- We loop using the given interaction as our list
            -- of continuations to execute.
            engineLoop opts c state i
```

The main loop of the engine is very simple. The first thing the function does is starting all tasks given as argument on lightweight threads. To do so, the function updates the number of running atoms and actually spawns lightweight threads for the different atoms using the `spawnAtom` function we will investigate next.

Then, the main loops waits for a notification from one of the spawned atom. As we will see, this notification is triggered by the last of the running atoms. Note that, for this reason, it is extremely important that the number of atoms is incremented before the atoms start executing, and not after, as atoms rely on this information to know if all other atoms have finished executing.

The next step performed by the engine, after being notified, is to compute a non-empty interaction from the waiting list. To do so, the closed semantics `getClosedInteractions` function is used. If no such interaction is possible, then the engine stops its execution. This situation can arise because:

- The waiting list is empty, and therefore there are no interactions to execute anymore. The system thus terminates normally.

- The waiting list is not-empty, but, even though there still are some atoms waiting, no non-empty interactions are possible. This situation represents a deadlock.

Finally, in the case when some interactions are possible, an interaction is chosen (by the function specified in option) to be executed. The corresponding atoms are then removed from the waiting list and the function loop backs using the tasks from the interaction.

The `spawnAction` function

The last piece of the puzzle is the `spawnAction` function, which is responsible to spawn a lightweight thread that will interpret the instructions that must be executed by the atom.

```
-- | Spawns an atom on its own lightweight thread.
--
--   The number of threads MUST be adapted accordingly before calls
--   to this function are made.
spawnAtom :: SystemState s -> Task s -> IO ()
spawnAtom state (Task a x) =
    void $ flip forkFinally terminateThread $ interpret $ getInstructions x
  where
    terminateThread _ = do
        -- We decrement the number of running atoms.
        n' <- decrementAndFetch (getRunningVar state)

        -- If there isn't any atom left running, we notify the engine.
        when (n' == 0) $ signal (getSemaphore state)

    interpret p = do
        -- Obtaining the next atom instruction.
        !v <- viewT p
        case v of
            Return _ -> do
                return ()  -- Done executing.

            Spawn x' :>>= f -> do
                -- We are instructed to create a new atom.

                -- Getting and updating the next identifier.
                i <- fetchAndIncrement (getNextIdVar state)

                -- Updating the number of running atoms.
                increment (getRunningVar state)

                -- Creating the identifier of the atom.
                let a' = AtomId i

                -- Spawn the new atom.
                spawnAtom state $ Task a' (void x')

                -- Execute the rest of the computation of the parent,
                -- with the identifier of the created atom made available.
                interpret $ f a'

            Await cs :>>= f ->
                -- We are instructed to wait on some ports.
```

```
                when (not $ null cs) $ do

                    -- Defines what to do for each single WaitCase.
                    let go w (AwaitCase p u g) =
                            setWaiting a p u (Action . f . g) w

                    -- Modify the waiting list to take
                    -- into account all WaitCase's.
                    w <- takeMVar (getWaitingListVar state)
                    let w' = foldl' go w cs
                    putMVar (getWaitingListVar state) w'

            GetId :>>= f -> do
                -- We are instructed to return the identifier of the atom.

                -- Simply gives the identifier
                -- to the rest of the computation.
                interpret $ f a
```

The first part of the function simply instructs the runtime system of Haskell to spawn a lightweight thread fot the interpreter of the atom. When the underlying lightweight thread terminates, normally or due to an exception, the `terminateThread` procedure is executed, which decrements the number of running threads and then notifies the main loop in case the counter reached 0.

The interpreter function of atoms simply looks at the sequence of instructions.

1. In the `Return _` case, the sequence of instructions is over, and the atom has thus finished its execution. The interpreter has nothing left to do.

2. In the `Spawn x' :>>= f` case, the first instruction of the sequence is a `spawn` instruction, followed by a function f which returns the rest of the computation. The interpreter has thus to create a new atom and feed its identifier to the rest of the computation.

   To do so, the identifier counter is fetched and incremented to obtain the identifier of the atom to spawn. The counter of running atoms is also incremented. Then the new atom is actually spawned and can begin its execution on a separate thread of execution. Finally, the rest of the instructions of the parent agent is obtained by feeding the identifier of the child atom to the function `f`, and then interpreted.

3. In the `Await cs :>>= f` case, the first instruction of the sequence is an `await` instruction, followed by a function f which returns the rest of the computation. Therefore, the interpreter has to register in the waiting list that the agent is waiting on the specified ports. Then, the interpreter finishes its execution. The atom will be able to restart once it is part of an interaction.

4. Finally, in the case of `GetId :>>= f`, the first instruction of the sequence is a `getSelfId` instruction, followed by a function f which returns the rest of the computation. The rest of the computation is simply obtained using by feeding the atom identifier to the function f. The rest of the instructions are then interpreted.

Final words

This concludes the exposition of the implementation of the BIP framework in Haskell. Throughout this section, we have seen in details how the conceptual constructs of BIP are actually implemented in the framework. In addition to be backed by a thorough mathematical formalisation, the implementation we have shown is small enough to be walked through in one go.

We will in the next section explore the Scala implementation of the framework. After that, we will discuss some optimisations that can be performed to speed up the execution of BIP systems. Before we conclude this thesis, we will also actually implement some example applications using both the Scala and Haskell framework.

4.3 Scala implementation

In this section we look at the implementation of BIP as a domain specific language in Scala. As in the discussion of the Haskell framework, we start from high level concepts of systems, atoms and ports down to the low level details of the engine.

4.3.1 Systems

The top level concept behind the framework is the concept of systems. A system regroups many atoms and ports and a single connector. In the Scala framework, a new empty system is obtained as follows.

```
val system = new System
```

The atoms and ports of a system are created using the methods `newAtom` and `newPort`, as exemplified below. The connector of the system is specified using the `registerConnector` method.

```
// Getting an empty system
val system = new bip.System

// Creating some ports
val port1 = system.newPort
val port2 = system.newPort

// Creating a first atom
val atom1 = system.newAtom {
  // Behaviour of the atom here
}

// Creating a second atom
val atom2 = system.newAtom {
  // Behaviour of the atom here
}
```

The above code do not start executing the system. Indeed, `System` is merely a builder class, which simply builds under the scene the initial configuration of a system. Since it is straightforwardly defined and quite verbose, we decided not to show its trivial implementation here.

4.3.2 Atom and ports

Atoms and ports in the framework only serve as identifiers. They are implemented as follows.

```scala
/** Represents an atom of a BIP system.
  *
  * @param identifier The identifier of the atom.
  */
case class Atom private (identifier: Int) {
  // ...
}

/** Companion object of [[bip.Atom]]. */
private object Atom {
  var _nextIdentifier = 0;

  /** Creates a new unique atom instance. */
  def newInstance(): Atom = {
    val identifier = synchronized {
      val i = _nextIdentifier
      _nextIdentifier += 1
      i
    }
    Atom(identifier)
  }
}
/** Represents a port of a BIP system.
  *
  * @tparam D The type of values that can be received by the port.
  * @tparam U The type of values that can be sent through the port.
  * @param identifier The identifier of the atom.
  */
case class Port[D, U] private (identifier: Int) {
  // ...
}

/** Companion object of [[bip.Port]]. */
private object Port {
  var _nextIdentifier = 0;

  /** Creates a new unique port instance. */
  def newInstance[D, U](): Port[D, U] = {
    val identifier = synchronized {
      val i = _nextIdentifier
      _nextIdentifier += 1
      i
    }
    Port(identifier)
```

```
  }
}
```

Since their constructor is private, the only way to obtain a port or atom is through the `newAtom` and `newPort` methods we have previously shown.

Note that the body of the classes are not shown yet. They uniquely contain methods that give the framework a more natural syntax. We will look at those methods when appropriate.

4.3.3 Atom actions

In addition to performing any arbitrary computation, atoms may perform any of the three following actions:

- Waiting on ports, using either the `await` or `awaitAny` functions. Those two instructions allow atoms to send and receive values from ports.
- Spawning new atoms, using the `spawnAtom` function. A reference to the newly created atom is returned.
- Getting a reference to the atom itself, using the `getSelf` function.

Note, and this is very important, that all the above functions are *not* methods of `Atom` or any other object. They are just top level functions. The atom executing those instruction is implicit, and is tightly related to the thread executing. As can be seen in the implementation of the functions shown below, the atom responsible for the execution of some actions can be recovered from the custom `Thread` executing. We will discuss this type of threads later when we look at the engine.

```
/** This package contains actions that can be performed by atoms.
  *
  * All of the functions defined below are meant to be used by atoms only.
  */
package object actions {

  /** Sends a value on potentially multiple ports, and waits for
    * any of them to receive a value back in return.
    *
    * @tparam A The type of values that can be received.
    * @param cases The different ports and values to send.
    * @param cont  What to do with the value received.
    *
    * @note This function never returns normally.
    */
  def awaitAny[A](cases: AwaitCase[_, _, A]*)(cont: A => Unit): Nothing = {
    // We throw a control exception containing all information we need.
```

```
    throw new AwaitException(cases, cont)
  }


  /** Sends a value on a port, and waits to receive a value back in return.
    *
    * @tparam D The type of values that can be received.
    * @tparam U The type of values that can be sent.
    * @param port  The port on which to wait.
    * @param value The value to be sent.
    * @param cont  What to do with the value received.
    *
    * @note This function never returns normally.
    */
  def await[D, U](port: Port[D, U], value: U)(cont: D => Unit): Nothing =
    awaitAny(AwaitCase(port, value, (x: D) => x))(cont)

  /** Sends a useless value on a port,
    * and waits to receive a value back in return.
    *
    * @tparam D The type of values that can be received.
    * @tparam U The type of values that can be sent.
    * @param port  The port on which to wait.
    * @param value The value to be sent.
    * @param cont  What to do with the value received.
    *
    * @note This function never returns normally.
    */
  def await[D](port: Port[D, Unit])(cont: D => Unit): Nothing =
    await(port, ())(cont)


  /** Spawns a new atom.
    *
    * @param action The action to execute.
    * @return The newly created atom.
    */
  def spawnAtom(action: => Unit): Atom = {
    // Gets the current thread.
    val thread: Thread = Thread.currentThread()

    // Ensures that the thread has been created by the engine.
    if (!thread.isInstanceOf[AtomExecutorThread]) {
      throw new Error("spawnAtom wasn't called from an atom.")
```

```
    }

    // Converts to the specific type of threads.
    val atomThread = thread.asInstanceOf[AtomExecutorThread]

    // Calls the method on the thread.
    atomThread.spawnAtom(() => action)
  }


  /** Returns the currently executing atom. */
  def getSelf(): Atom = {
    // Gets the current thread.
    val thread: Thread = Thread.currentThread()

    // Ensures that the thread has been created by the engine.
    if (!thread.isInstanceOf[AtomExecutorThread]) {
      throw new Error("getSelf wasn't called from an atom.")
    }

    // Converts to the specific type of threads.
    val atomThread = thread.asInstanceOf[AtomExecutorThread]

    // Gets the atom being executed by the thread.
    atomThread.atom
  }
}
```

The functions presented above are simply there for syntactic purposes. The real work will be performed by the thread executing the behaviour of the atom, as we will see in the section on the engine. The thread is either involved via a method call, in the case of `spawnAtom` and `getSelf`, or via an exception, in the case of the different versions of the `await` instruction.

Also note that the return type of `await` and `awaitAny` is `Nothing`, which signifies that the function can not return normally. Indeed, we do not want to block the underlying thread on potentially very long `await` instructions. For this reason, the continuation of the atom must be explicitly passed as an argument.

`AwaitCase`

In the previous code snippet, you may have notice the use of the `AwaitCase` class in the signature of the `awaitAny` function. Values of type `AwaitCase` are simply glorified port-value pairs.

```
/** Contains a port and a value to be sent on that port.
  * Also contains a function to apply on the received value.
  *
  * @tparam D The type of values that can be received by the port.
  * @tparam U The type of values that can be sent by the port.
  * @tparam A The type of the received value, after being transformed.
  * @param port     The port on which to send a value.
  * @param value    The value to send.
  * @param function The function to apply on the received value.
  * @see See the method [[Port.withValue]] to create values of this type.
  */
case class AwaitCase[D, U, A] private[bip] (
    port: Port[D, U],
    value: U,
    function: D => A) {

  /** Chains another function to be applied on the received value. */
  def map[B](other: A => B): AwaitCase[D, U, B] =
    AwaitCase(port, value, function andThen other)
}
```

The only way for the users of the framework to obtain a value of type `AwaitCase` is to use the method `withValue` from the `Port` class. This touch gives the DSL a more natural looking syntax.

```
awaitAny(port1 withValue value1, port2 withValue value2) {
  // What to do when a value is received.
}
```

Where the `withValue` method is defined as:

```
case class Port[D, U] private (identifier: Int) {

  /** Returns an object that specifies that the given value
    * should be sent on this port.
    */
  def withValue(value: U): AwaitCase[D, U, D] =
    AwaitCase(this, value, (x: D) => x)
}
```

4.3.4 Connectors

So far, all the classes and methods we have seen were in some sense the interface of the framework for the Behaviour layer of the system. Apart from performing some trivial transformations and propagating method calls, the code we have seen did not perform

much. Before we can look at the implementation of the engine, which will actually handle those calls and run the system, we have the Glue layer to investigate.

One of the main concept of the Glue layer is the concept of connectors. In the framework, the connectors are implemented as an algebraic data type. This implementation is very close to the formalisation of connector combinators we have seen in the previous chapters.

The `Connector` type

Before we look at the different case classes, let us spend some time on the type of the connectors. The methods of the class will be discussed later.

```
/** Connects together the ports of a system.
  *
  * @tparam D The type of values accepted during the downwards phase.
  * @tparam U The type of values propagated during the upwards phase.
  */
sealed abstract class Connector[-D, +U] {
  // ...
}
```

A you can see, connectors are parameterised by two types: `D`, which is the type of values that are accepted by the connector during the downwards phase, and `U`, which is the type of values that are propagated during the upwards phase.

Note that the two type parameters are given variance annotations. The type parameter `U` is in *covariant* position, meaning that if the type `U` is a subtype of some type `V`, then connectors of type `Connector[D, U]` are also subtypes of `Connector[D, V]`. Intuitively, a connector that propagates upwards values of a certain type also propagate upwards values of all its super types.

To the contrary, `D` is in *contravariant* position, meaning that if the type `D` is a *super type* of some type `E`, then the connectors of type `Connector[D, U]` are *subtypes* of `Connector[E, U]`. Indeed, a connector that accepts values of a certain type during the upwards phase also accepts values that are subtypes of that type.

Core combinators

Now that we have discussed the type of connectors, we can look at the different case classes that are subtypes of `Connector`.

For now, we will not look at the methods of connectors, but simply at the different case classes available. As you can see, the case classes each correspond to one of the different connector combinator we have formalised.

```
// Core combinators

/** Binds a port to an atom. */
```

```
private case class Bind[D, U](
    atom: Atom,
    port: Port[D, U]) extends Connector[D, U] {
  // ...
}

/** Successful connector always propagating the same upwards value. */
private case class Success[U](value: U) extends Connector[Any, U] {
  // ...
}

/** Failing connector, never enabled. */
private case object Failure extends Connector[Any, Nothing] {
  // ...
}

/** Synchronization of two connector. */
private case class BothOf[D, U, V](
    left: Connector[D, U],
    right: Connector[D, V]) extends Connector[D, (U, V)] {

   // ...
}

/** Union of two connectors. */
private case class OneOf[D, U](
    left: Connector[D, U],
    right: Connector[D, U]) extends Connector[D, U] {
  // ...
}
```

Data combinators

```
// Data combinators

/** Applies a function on the upwards value. */
private case class Mapped[D, U, V](
    function: U => V,
    connector: Connector[D, U]) extends Connector[D, V] {
  // ...
}

/** Applies a function on the downwards value. */
private case class ContraMapped[D, E, U](
```

```
    function: E => D,
    connector: Connector[D, U]) extends Connector[E, U] {
  // ...
}

/** Ensures that a predicate holds on the upwards value. */
private case class Guarded[D, U](
    predicate: U => Boolean,
    connector: Connector[D, U]) extends Connector[D, U] {
  // ...
}

/** Propagates the upwards value back during the downwards phase. */
private case class Feedback[D, U](
    connector: Connector[(D, U), U]) extends Connector[D, U] {
  // ...
}
```

Priority combinators

```
// Priority combinators

/** Gives priority to a connector. */
private case class FirstOf[D, U](
    left: Connector[D, U],
    right: Connector[D, U]) extends Connector[D, U] {
  // ...
}

/** Ensures that upwards values are maximal according to some ordering. */
private case class Maximal[D, U](
    ordering: PartialOrdering[U],
    connector: Connector[D, U]) extends Connector[D, U] {
  // ...
}
```

Dynamic combinator

```
// Dynamic combinators

/** Accepts any atom on the given port. */
private case class Dynamic[D, U](port: Port[D, U]) extends Connector[D, U] {
  // ...
}
```

```scala
/** Synchronizes with the connector provided as upwards value. */
private case class Joined[D, U](
    connector: Connector[D, Connector[D, U]]) extends Connector[D, U] {
  // ...
}
```

The above representation of connectors as an algebraic data type is very similar to the formalisation we have seen of the connector combinators. This representation of connectors as a tree-like structure will allow us to manipulate the connectors using pattern matching. Note that the semantics of the connectors has not yet been shown. What we have here is in some sense a typed grammar for the connectors.

Library of connector combinators

As you may have noticed, the case classes are declared `private`. This means that users of the framework will not be exposed to the connector combinators directly. Instead, they are given a library of function from which they can build their own connectors. This also has the advantage that the API offered by the framework does not depend on the internal implementation of connectors. Some of the functions provided, and their implementation, are presented below. The unary and binary combinators are implemented as methods of the `Connector` class. On the other hand, the n-ary versions of the combinators are to be found in a separate object.

Unary and binary combinators

```scala
sealed abstract class Connector[-D, +U] {

  /** Applies a function on the upwards value. */
  def map[V](function: U => V): Connector[D, V] = Mapped(function, this)

  /** Applies a function on the downwards value. */
  def contramap[E](function: E => D):
      Connector[E, U] = ContraMapped(function, this)

  /** Applies a function on the upwards value and synchronize
    * with the connector produced by the function.
    */
  def flatMap[V >: U, E <: D](function: U => Connector[E, V]):
      Connector[E, V] = Joined(Mapped(function, this))

  /** Ensures that a predicate holds on the upwards value. */
  def filter(predicate: U => Boolean):
      Connector[D, U] = Guarded(predicate, this)
```

```
/** Ensures that a predicate holds on the upwards value. */
def withFilter(predicate: U => Boolean):
    Connector[D, U] = Guarded(predicate, this)

/** Specifies an alternative to this connector. */
def or[E <: D, V >: U](that: Connector[E, V]): Connector[E, V] =
  OneOf(this, that)

/** Synchronization of two connectors. Propagates upwards both values. */
def and[E <: D, V](that: Connector[E, V]): Connector[E, (U, V)] =
  BothOf(this, that)

/** Synchronization of two connectors. Propagates upwards the left value. */
def andLeft[E <: D](that: Connector[E, Any]): Connector[E, U] = {
  def first(t: (U, Any)) = t._1
  Mapped(first, BothOf(this, that))
}

/** Synchronization of two connectors. Propagates upwards the right value. */
def andRight[E <: D, V](that: Connector[E, V]): Connector[E, V] = {
  def second(t: (U, V)) = t._2
  Mapped(second, BothOf(this, that))
}

/** Replaces the upwards value. */
def sending[V](value: V): Connector[D, V] =
  Mapped((x: U) => value, this)

/** Replaces the downwards value. */
def receiving(value: D): Connector[Any, U] =
  ContraMapped((x: Any) => value, this)

/** Provides the upwards value to the downwards function. */
def feedback[E](implicit function: (E, U) => D): Connector[E, U] = {
  val uncurried = (t: (E, U)) => function(t._1, t._2)
  val connector: Connector[(E, U), U] = this.contramap(uncurried)
  Feedback(connector)
}

/** Uses the upwards value as the downwards value.
  * Ignores all downwards values it receives.
  * Propagates upwards only the unit value.
  */
def close(implicit function: U => D): Connector[Any, Unit] = {
```

```scala
    val second = (t: (Any, U)) => t._2
    Feedback(this.contramap(second andThen function)).sending(())
  }

  // ...
}
```

N-ary combinators

```scala
package object connectors {

  /** Binds an atom to a port. */
  def bind[D, U](atom: Atom, port: Port[D, U]): Connector[D, U] =
    Bind(atom, port)

  /** Always provides the same upwards value. */
  def success[U](value: U): Connector[Any, U] = Success(value)

  /** Never provide any upwards value. */
  val failure: Connector[Any, Nothing] = Failure

  /** Disjunction of connectors. */
  def anyOf[D, U](connectors: Connector[D, U]*): Connector[D, U] = {
    val zero: Connector[D, U] = failure
    connectors.foldRight(zero)(OneOf(_, _))
  }

  /** Synchronization of connectors. */
  def allOf[D, U](connectors: Connector[D, U]*): Connector[D, Seq[U]] = {
    val one: Connector[D, Seq[U]] = success(Seq())
    def concat(t: (U, Seq[U])): Seq[U] = t._1 +: t._2
    connectors.foldRight(one) {
      case (first, rest) => Mapped(concat, BothOf(first, rest))
    }
  }

  /** Synchronization of any number of underlying connectors. */
  def manyOf[D, U](connectors: Connector[D, U]*): Connector[D, Seq[U]] = {
    allOf(connectors.map(_.map(Seq(_)).or(success(Seq()))) : _*).map(_.flatten)
  }

  /** Disjunction of connectors with priority. */
  def firstOf[D, U](connectors: Connector[D, U]*): Connector[D, U] = {
    val zero: Connector[D, U] = failure
```

```
    connectors.foldRight(zero)(FirstOf(_, _))
  }

  /** Binds any atom on the port. */
  def dynamic[D, U](port: Port[D, U]): Connector[D, U] = Dynamic(port)

  /** Uses the upwards value as the downwards value. */
  def close[D, U <: D](connector: Connector[D, U]): Connector[Any, Unit] = {
    val second = (t: (Any, U)) => t._2
    Feedback(connector.contramap(second)).sending(())
  }
}
```

It is important to note that the small library of combinators that we have shown could be augmented with many other derived combinators, either by the framework or its users.

4.3.5 Interactions

At this point in the exposition of the framework, we have seen all of the aspects concerning the user-facing domain specific language. In the rest of this section, we will walk through the implementation of the backend of the Scala framework. The first thing we need to show is how the interactions are encoded within the framework. As in the formalisation, we make the distinction between closed and open interactions.

```
/** Open interaction of a system.
  *
  * @tparam D The type of the value needed in the downwards phase.
  * @tparam U The type of the value provided in the upwards phase.
  * @param upwards   The value transmitted during the upwards phase.
  * @param downwards The function that computes the action to
  *                  be executed by each of the involved atoms.
  * @param involved  The atoms involved in the interaction.
  */
private case class OpenInteraction[-D, +U](
    upwards: U,
    downwards: D => Seq[(Atom, () => Unit)],
    involved: Set[Atom])


/** Closed interaction of a system.
  *
  * @param tasks A list of actions to be performed by atoms.
  */
```

```
private case class Interaction(tasks: Seq[(Atom, () => Unit)])


/** Companion  object of [[bip.Interaction]]. */
private object Interaction {

  /** Closes an open interaction and produces a closed interaction. */
  def close[A](open: OpenInteraction[A, A]): Interaction = open match {
    case OpenInteraction(upwards, downwards, _) =>
      Interaction(downwards(upwards))
  }
}
```

As we will see, interactions will be created by the semantic function which, given a connector, returns a stream of all possible interactions in the current system state. But first, we need to have a look at the data structure that contains all information on waiting atoms. This data structure is the waiting list.

4.3.6 Waiting list

The waiting list of a system is a data structure that contains an entry for each pair of atom and port, such that the atom is waiting on the port. As we have already seen, waiting lists support the following operations:

- Registering an atom as waiting on a specific port.
- Removing all entries related to a given atom. This specifies that the atom is now running and thus no longer waiting on any port.
- Getting the open interaction, if any, corresponding to an atom and port.
- Getting all open interactions involving a port.

In Scala, the data structure and operations on it are implemented as follows:

```
/** Contains all information about waiting atoms. */
private class WaitingList {

  /** Main mapping, contains all entries */
  val portsForAtom: HashMap[Atom, HashMap[Port[_, _], WaitCase[_, _]]] =
    HashMap.empty

  /** Secondary mapping, contains all atoms waiting on a specific port. */
  val atomsForPort: HashMap[Port[_, _], LinkedHashSet[Atom]] = HashMap.empty
```

```
/** Records that the atom is waiting on the specified port.
  *
  * @param atom The waiting atom.
  * @param port The activated port.
  * @param sent The value sent by the atom through the port.
  * @param cont The continuation of the atom.
  */
def setWaiting[D, U](
    atom: Atom,
    port: Port[D, U],
    sent: U,
    cont: D => Unit) {

  val set = atomsForPort.getOrElseUpdate(port, LinkedHashSet.empty)
  set += atom

  val map = portsForAtom.getOrElseUpdate(atom, HashMap.empty)
  map(port) = WaitCase(sent, cont)
}

/** Specifies that the atom is no longer waiting.
  *
  * Removes all entries related to the atom in the data structure.
  *
  * @param atom The atom that is no longer waiting.
  */
def setRunning(atom: Atom) {

  val binding = portsForAtom.remove(atom)

  binding match {
    case Some(ports) => ports.keys.foreach {
      case port => {
        val set = atomsForPort(port)
        set -= atom
        if (set.isEmpty) {
          atomsForPort -= port
        }
      }
    }
    case None => ()  // Atoms was already running.
  }
}
```

```
  /** Returns the entry related to the atom and port. */
  def getBoundInteraction[D, U](atom: Atom, port: Port[D, U]):
      Option[OpenInteraction[D, U]] = for {
    ports <- portsForAtom.get(atom)
    untypedWaitCase <- ports.get(port)
  } yield {
    val waitCase: WaitCase[D, U] = untypedWaitCase.asInstanceOf[WaitCase[D, U]]

    OpenInteraction(
      waitCase.upwards,
      (x: D) => Seq((atom, () => waitCase.downwards(x))),
      Set(atom))
  }

  /** Returns all entries related to the port. */
  def getBoundInteractions[D, U](port: Port[D, U]):
      Stream[OpenInteraction[D, U]] = {
    val atoms = atomsForPort.getOrElse(port, LinkedHashSet.empty)

    val zero: Seq[OpenInteraction[D, U]] = Seq()

    atoms.foldRight(zero)({
      case (atom, rest) => getBoundInteraction(atom, port) match {
        case None => rest
        case Some(interaction) => interaction +: rest
      }
    }).toStream
  }
}

/** Entry of the [[bip.WaitingList]]. */
private case class WaitCase[D, U](upwards: U, downwards: D => Unit)
```

4.3.7 Semantic function

Has we have just shown the implementation of the waiting list, we are in position to show how the semantic function defined during the formalisation can be implemented in Scala. The semantic function is implemented as a method of the `Connector` class.

```
sealed abstract class Connector[-D, +U] {
  // ...

  /** Returns a stream of all currently possible interactions.
```

```
    *
    * @param waitingList The data structure holding information
    *                    about all waiting atoms.
    */
  private[bip] def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]]
}
```

Each primitive combinator overrides this method and implements the related part of the semantic function.

```
// Core combinators

private case class Bind[D, U](
    atom: Atom,
    port: Port[D, U]) extends Connector[D, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]] = {

    waitingList.getBoundInteraction(atom, port).toStream
  }
}

private case class Success[U](value: U) extends Connector[Any, U] {
  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[Any, U]] = {

    Stream(OpenInteraction(value, (x: Any) => Seq(), Set()))
  }
}

private case object Failure extends Connector[Any, Nothing] {
  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[Any, Nothing]] = Stream()
}

private case class BothOf[D, U, V](
    left: Connector[D, U],
    right: Connector[D, V]) extends Connector[D, (U, V)] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, (U, V)]] = for {
    OpenInteraction(u1, d1, is1) <- left.getInteractions(waitingList)
```

```
    OpenInteraction(u2, d2, is2) <- right.getInteractions(waitingList)
    is3 <- Stream(is1 union is2)
    if (is3.size == is1.size + is2.size)
  } yield OpenInteraction((u1, u2), (x: D) => d1(x) ++ d2(x), is3)
}

private case class OneOf[D, U](
    left: Connector[D, U],
    right: Connector[D, U]) extends Connector[D, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]] = {

    left.getInteractions(waitingList) ++ right.getInteractions(waitingList)
  }
}

// Data combinators

private case class Mapped[D, U, V](
    function: U => V,
    connector: Connector[D, U]) extends Connector[D, V] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, V]] = {

    connector.getInteractions(waitingList) map {
      case OpenInteraction(upwards, downwards, involved) =>
        OpenInteraction(function(upwards), downwards, involved)
    }
  }
}

private case class ContraMapped[D, E, U](
    function: E => D,
    connector: Connector[D, U]) extends Connector[E, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[E, U]] = {

    connector.getInteractions(waitingList) map {
      case OpenInteraction(upwards, downwards, involved) =>
        OpenInteraction(upwards, function andThen downwards, involved)
    }
```

```
  }
}

private case class Guarded[D, U](
    predicate: U => Boolean,
    connector: Connector[D, U]) extends Connector[D, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]] = {

    connector.getInteractions(waitingList) filter {
      case OpenInteraction(upwards, _, _) => predicate(upwards)
    }
  }
}

private case class Feedback[D, U](
    connector: Connector[(D, U), U]) extends Connector[D, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]] = {

    connector.getInteractions(waitingList) map {
      case OpenInteraction(upwards, downwards, involved) =>
        OpenInteraction(upwards, (x: D) => downwards((x, upwards)), involved)
    }
  }
}

// Priority combinators

private case class FirstOf[D, U](
    left: Connector[D, U],
    right: Connector[D, U]) extends Connector[D, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]] = {

    val lefts = left.getInteractions(waitingList)

    if (!lefts.isEmpty) {
      lefts
    }
    else {
```

```
      right.getInteractions(waitingList)
    }
  }
}

private case class Maximal[D, U](
    ordering: PartialOrdering[U],
    connector: Connector[D, U]) extends Connector[D, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]] = {

    val interactions = connector.getInteractions(waitingList).toList

    val upwardsValues = interactions.map(_.upwards)

    def maximal(o: OpenInteraction[D, U]): Boolean = o match {
      case OpenInteraction(upwards, _, _) =>
        !upwardsValues.exists(ordering.gt(_, upwards))
    }

    interactions.filter(maximal).toStream
  }
}

// Dynamic combinators

private case class Dynamic[D, U](port: Port[D, U]) extends Connector[D, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]] = waitingList.getBoundInteractions(port)

}

private case class Joined[D, U](
    connector: Connector[D, Connector[D, U]]) extends Connector[D, U] {

  override def getInteractions(waitingList: WaitingList):
      Stream[OpenInteraction[D, U]] = for {
    OpenInteraction(u1, d1, is1) <- connector.getInteractions(waitingList)
    OpenInteraction(u2, d2, is2) <- u1.getInteractions(waitingList)
    is3 <- Stream(is1 union is2)
    if (is3.size == is1.size + is2.size)
  } yield OpenInteraction(u2, (x: D) => d1(x) ++ d2(x), is3)
```

```
}
```

4.3.8 Engine

Now that we have seen how the semantic function is implemented in Scala, we can have a look at the how the engine executing the system is implemented. As we have previously discussed, the engine is composed of two distinct active parts: the main loop and the worker threads. We start our exposition by showing the implementation of the main loop.

Main loop

As we previously explained, the main loop executes repeatedly the same series of actions. First of all, the loop creates the worker threads to execute the current batch of tasks. Then, the loop waits for the engine to reach a stable state. Then, once a stable state has been reached, an interaction is taken from the semantic function and the loops starts over using the current interaction as the list of tasks to execute. It is implemented as follows:

```
/** Executes a BIP system.
  *
  * @param closed       The connector of the system.
  * @param initialTasks The initial tasks to execute
  *                     at the beginning of the system.
  * @param options      User specified options.
  */
private class Engine(
    closed: ClosedConnector[_],
    initialTasks: Seq[(Atom, () => Unit)],
    val options: SystemOptions) {

  /** Waiting list of the system. */
  private val _waitingList = new WaitingList

  /** Number of currently running atoms. */
  private var _running = 0

  /** Indicates whether a worker thread has signaled the main loop. */
  private var _signaled = false

  /** Executes the engine. */
  def run() {

    // Tasks to execute.
```

```
var tasks = initialTasks

// Main loop of the engine.
while (true) {

  // If there are some tasks to execute,
  // start the worker threads.
  if (!tasks.isEmpty) {
    modifyRunningCount(tasks.length)

    tasks.foreach {
      case (atom, action) => {
        spawnAtom(atom, action)
      }
    }
  }
  tasks = Seq()

  // Wait for the main loop to be signaled.
  synchronized {
    if (!_signaled) {
      wait()
    }
    assert(_signaled)
    _signaled = false
  }

  // Computes the next interaction.
  val interactionsStream = closed.getInteractions(_waitingList)

  val nonEmptyInteractionsStream =
    interactionsStream.filter(!_.tasks.isEmpty)

  // Checks if an interaction is possible.
  if (!nonEmptyInteractionsStream.isEmpty) {

    // Picks an interaction.
    tasks = options.interactionPicker(nonEmptyInteractionsStream).tasks

    // Removes the involved atoms from the waiting list.
    tasks.foreach {
      case (atom, _) => _waitingList.setRunning(atom)
    }
  }
```

```
    else {
      // Not a single possible interaction.
      return
    }
  }
}

// Functions below are called by worker threads.

/** Modifies the number of running atoms. */
def modifyRunningCount(delta: Int): Int = synchronized {
  _running += delta
  _running
}

/** Signals the main loop that it main continue its execution. */
def signal() {
  synchronized {
    _signaled = true
    notify()
  }
}

/** Creates and starts a worker thread.
  *
  * @param atom   The atom whose action is executed.
  * @param action The action to be performed by the atom.
  */
def spawnAtom(atom: Atom, action: () => Unit) {
  val thread = new AtomExecutorThread(this, atom, action)
  thread.start()
}


/** Registers in the waiting list of the engine that the
  * atom is waiting on the given ports.
  *
  * @param atom  The atom that is declared as waiting.
  * @param cases The different ports and associated values.
  * @param cont  The continuation of the atom.
  */
def setWaiting[A](
    atom: Atom,
    cases: Seq[AwaitCase[_, _, A]],
```

```
      cont: A => Unit) {

    _waitingList.synchronized {
      cases.foreach {
        case AwaitCase(port, value, function) =>
          _waitingList.setWaiting(atom, port, value, function andThen cont)
      }
    }
  }
}
```

In this particular implementation, we have made heavy use of the synchronisation primitives offered by Scala and the JVM. In particular, mutual exclusion is enforced using `synchronized` blocks. All accesses made to variables shared on multiple thread is always performed in a `synchronized` block of the engine, which also ensures the memory visibility of all operations. When the main loop needs to wait, it uses the `wait()` function of the underlying monitor[4]. As we will shortly see, the main loop will be able to resume its execution when the underlying monitor gets notified by a worker thread.

Worker thread

The last point of the implementation we have left to see is the definition of the worker thread class. As previously discussed, its goal is to run the behaviour of an atom and handle all instruction calls made by the atom.

```
/** Worker thread of the engine.
  *
  * @param engine The engine which created the thread.
  * @param atom   The atom whose behaviour is being executed.
  * @param action The action that must performed by the atom.
  */
private class AtomExecutorThread(
    engine: Engine,
    val atom: Atom,
    action: () => Unit) extends Thread {

  /** Runs the action and handles all calls to atom instructions. */
  override def run() {
    try {
      // Try executing the action.
      action()
    }
    catch {
      // Handling calls to await and awaitAny.
```

```
      case AwaitException(cases, cont) =>
        engine.setWaiting(atom, cases, cont)
    }
    finally {
      // Modifying the number of currently executing atoms.
      val running = engine.modifyRunningCount(-1)

      // Signaling the main loop if necessary.
      if (running == 0) {
        engine.signal()
      }
    }
  }

  /** Creates a new atom and returns its identifier. */
  def spawnAtom(action: () => Unit): Atom = {
    engine.modifyRunningCount(1)
    val newAtom = Atom.newInstance()
    engine.spawnAtom(newAtom, action)
    newAtom
  }
}

/** Control exception thrown by calls to [[bip.actions.await]]
  * or [[bip.actions.awaitAny]].
  *
  * Will be caught by the [[bip.AtomExecutorThread]] executing the atom.
  *
  * @param cases The different values sent on the different ports.
  * @param cont  The continuation of the atom.
  */
private case class AwaitException[A](
    cases: Seq[AwaitCase[_, _, A]],
    cont: A => Unit) extends Exception with ControlThrowable
```

The exposition of the definition of the worker thread class conclude the discussion of the Scala implementation of the framework. Throughout this section, we have seen in some details how the different aspects of the frameworks are implemented in Scala. Starting from the user-facing domain specific language down to the engine, we discussed every aspect of the implementation.

In the next chapter, we will discuss some optimisations that can be applied to obtain performance improvements. Even though the implementation of the optimisations will be presented in Haskell, they can most of the time be applied in the same manner in Scala.

Chapter 5

Optimisations

We have seen in the previous chapters the implementation of the concepts from BIP in both Haskell and Scala. The implementations we have shown are left intentionally simple, so that they stay easily comprehensible. We will however throughout this section discuss some optimisations that we have implemented and some others that might be implemented as future work. Since optimisations can generally be applied in the same manner to either the Scala or the Haskell framework, we will only show the implementations in Haskell.

Most of the optimisations we present will focus on speeding up the execution of the semantic function. Since we have proven that even knowing whether or not the semantic function return at least an interaction is *NP-hard*, we can not hope to get good performance on all cases. We will however try to highlight some cases where the performance can easily made better. By this, we hope to make the function run fast in typical situations.

5.1 Early detection of downwards incompatibility

The first optimisation we explore focuses on the semantics function. One huge performance improvement we were able to bring to this function was obtained by the possibility to fail earlier in cases when it is clear that some of the interactions produced will be discarded later. As we will see, this is particularly useful in the case of connectors making heavy use of synchronisation combinators such as `BothOf` and `Joined`. As we have seen, in those connectors, interactions are combined to get larger interactions. However, interactions can only be combined with others if they are downwards compatible, that is if they do not involve the same atoms. In some cases, this downwards compatibility check leads to a very large amount of interactions being discarded, and thus a very large amount of time wasted creating those interactions for nothing.

The basic idea behind this optimisation is to avoid producing downwards incompatible interactions. For instance, in the case of `BothOf` c_1 c_2, instead of waiting for an interaction to be completely produced from c_2 before checking that it is downwards compatible with an interaction produced by c_1, we instruct the semantics not produce any interaction involving atoms previously used when inspecting c_2.

Before we propose a solution to this problem, let us expose in more details the issue with the current implementation of the semantic function.

Problematic cases

To better understand the problem, consider the following symptomatic connector:

$$\texttt{Problematic}_n := \texttt{AllOf} <\texttt{Dynamic}\ p \mid i \in [1, n]>$$

This connector takes the conjunction of n times the same dynamic connector, where we will assume n to be greater or equal to 2. It can straightforwardly be formulated in Haskell as:

```
problematic n p = allOf [ Dynamic p | i <- [ 1 .. n ] ]
```

Which is strictly equivalent to:

```
BothOf (Mapped (:) (Dynamic p)) (problematic (n - 1) p)
```

Imagine that there are at this point in time n different atoms $a_1 \ldots a_n$ waiting on the port `p`. In this case, the semantic function will return for `Mapped (:)  (Dynamic p)` a list of n interactions $o_1 \ldots o_n$, in some specific order. Without loss of generality, we will assume that the interaction at the position i in the list, o_i, involves the unique atom a_i.

The crucial part of the problem is in the way the semantics function works for `BothOf` (and `Joined`). When fed `BothOf` c_1 c_2, the function first fixes its choice of the interaction from the left-hand side c_1 and then exhaustively tries to get a downwards compatible interaction from the right-hand side c_2. Once this is completely done, the

function does the same exploration using the next interaction from the left-hand side c_1, and so on.

The problem is that, since the order of interactions coming from any of the `Mapped (:) (Dynamic p)` is fixed, all subparts will fix their choice to the same subsets of interactions, and thus return at the head of the list only downwards incompatible interactions.

For instance, to obtain the first of the possible interactions of `problematic n p`, the semantic function will first fix its choice of an interaction from the left-hand side. In this case, o_1 is obtained, since it is the first interaction returned from the left-hand side. The function then tries to pick an interaction from the right-hand side of `problematic n p`, without ever changing its choice for the left-hand side before the right-hand side is completely explored. Recall that the right-hand side is defined as:

```
problematic (n - 1) p
```

Applied to `problematic (n - 1) p`, the recursive call will be able to return $(n-1)!$ interactions. Due to the behaviour we have just exposed, at least the $(n-2)!$ (that is $n-2$ factorial) first of them will involved the atom a_1, since the choice of o_1 was fixed for the entire beginning of the execution.

This means that the semantic function will have to go through at least $(n-2)!$ downwards incompatible functions before finding a downwards compatible one. When n gets large, this very rapidly becomes computationally infeasible to go through that many candidate interactions.

However, the problem doesn't stop here! The same kind of complexity arises at each layer down the right-hand side of the `problematic` connector, leading to a massive explosion of the complexity. Below is displayed the running time of the `getInteractions` function on the `problematic` connector, for various sizes of n. Note that the running time displayed here are just samples, taken in the Haskell interpreter, on a personal laptop. They thus bear absolutely no statistical significance whatsoever. They are simply used for illustrative purposes.

Number of atoms n	Running time
1	< 0.1ms
2	< 0.1ms
3	~ 0.2ms
4	~ 0.6ms
5	~ 3.4ms
6	~ 27.7ms
7	~ 314.6ms
8	~ 5388.4ms
9	~ 102990.9ms
10	`Interrupted`

Proposed solution

As we have seen, the exposition in complexity was due to the fact that the semantic function tirelessly tried to return interactions that were, due to downwards incompatibility, not usable in the context they were needed.

This massive explosion in complexity can be avoided, in this specific case, by specifying directly to the semantic function which sets of atoms not to use. This will ensures that the semantic function does not spend time computing interactions that will be disregarded later on anyway due to downwards incompatibility. This modification of the semantic function is implemented as follows:

```haskell
-- | Gets the list of possible interactions. The list is computed lazily,
--   element by element.
getInteractions :: Connector s d u -> WaitingList s -> [OpenInteraction s d u]
getInteractions c w = getInteractionsWithout Set.empty c w

-- | Gets the list of possible interactions that do not involve
--   a specific set of atoms. The list is computed lazily,
--   element by element.
getInteractionsWithout :: Set (AtomId s) -> Connector s d u -> WaitingList s
                       -> [OpenInteraction s d u]
getInteractionsWithout as c w = case c of
    -- Core combinators
    Success x    -> [OpenInteraction x (const []) (Set.empty)]
    Failure      -> []
    Bind a p     -> if Set.member a as
        then []
        else maybeToList $ getBoundInteraction a p w
    OneOf c1 c2  ->
        getInteractionsWithout as c1 w ++
        getInteractionsWithout as c2 w
    BothOf c1 c2 -> do
        o1 <- getInteractionsWithout as c1 w
        -- We specify in the recursive call that using atoms involved in the
        -- open interaction o1 is prohibited.
        o2 <- getInteractionsWithout (Set.union as $ getInvolved o1) c2 w
        let o3 = OpenInteraction
                { getUpwards   = (getUpwards o1, getUpwards o2)
                , getDownwards = \ x -> getDownwards o1 x ++ getDownwards o2 x
                , getInvolved  = Set.union (getInvolved o1) (getInvolved o2)
                }
        return o3

    -- Data combinators
    Mapped f c1 -> do
```

```
        o1 <- getInteractionsWithout as c1 w
        return $ o1 { getUpwards = f (getUpwards o1) }
    ContraMapped f c1 -> do
        o1 <- getInteractionsWithout as c1 w
        return $ o1 { getDownwards = getDownwards o1 . f }
    Guarded f c1 -> do
        o1 <- getInteractionsWithout as c1 w
        guard (f $ getUpwards o1)
        return o1
    Feedback c1 -> do
        o1 <- getInteractionsWithout as c1 w
        return $ o1 { getDownwards =
            \ x -> getDownwards o1 (x, getUpwards o1) }

    -- Priority combinators

    -- Note that the first recursive call in the case of FirstOf and Maximal
    -- is made with an empty set of prohibited atoms. This is by design.
    -- It is not because an interaction might be considered downwards
    -- incompatible later on that it should loose its priority here.
    FirstOf c1 c2 -> case getInteractionsWithout Set.empty c1 w of
        []  -> getInteractionsWithout as c2 w
        os1 -> mfilter (not . any (flip Set.member as) . getInvolved) os1
    Maximal f c1 ->
        let os1 = getInteractionsWithout Set.empty c1 w
            upwardsValues = fmap getUpwards os1
            biggerThan x = \ y -> f y x == GT
            isMaximal x = not $ any (biggerThan x) upwardsValues
        in
        filter (isMaximal . getUpwards) $
        filter (not . any (flip Set.member as) . getInvolved) os1

    -- Dynamic combinators
    Dynamic p -> do
        o1 <- getBoundInteractions p w
        -- Filter out any interaction that involves an prohibited atom.
        guard $ not $ any (flip Set.member as) (getInvolved o1)
        return o1
    Joined c1 -> do
        o1 <- getInteractionsWithout as c1 w
        -- We specify in the recursive call that using atoms involved in the
        -- open interaction o1 is prohibited.
        o2 <- getInteractionsWithout (Set.union as $ getInvolved o1)
                (getUpwards o1) w
```

```
    let o3 = OpenInteraction
            { getUpwards   = getUpwards o2
            , getDownwards = \ x -> getDownwards o1 x ++ getDownwards o2 x
            , getInvolved  = Set.union (getInvolved o1) (getInvolved o2)
            }
    return o3
```

The `getInteractionsWithout` function is very similar to the `getInteractions` we have seen before. This function, compared to the previous one, takes a set of prohibited atom identifiers as an extra argument. In most cases, this argument is unused and simply passed untouched to the recursive calls. There are notable exceptions:

- In the case of `Bind` and `Dynamic`, the set of prohibited atoms is used to filter out interactions returned from the waiting list that might involve one of the forbidden atom.

- In the case of `BothOf` and `Joined`, the set of prohibited atoms is enriched with the downwards incompatible atoms before the recursive calls to the right-hand connector, or the connector upwards value in the case of `Joined`, are made.

  Also note that the check for disjointness is no longer necessary, since by constructions interactions will be compatible.

- Finally, special attention must be given to the priority combinators. Indeed, even though some interactions might be considered downwards incompatible later on, they potentially still have priority over other interactions at this point. For this reason, some recursive calls in the `FirstOf` and `Maximal` cases are made with an empty set of prohibited atoms. Then, the returned streams of interactions are filtered to remove invalid interactions.

Using this simple optimisation, the running time of the function is substantially reduced in some cases. Back to our symptomatic case, we now observe the following running time:

Number of atoms n	Previous running time	New running time
1	$<$ 0.1ms	$<$ 0.1ms
2	$<$ 0.1ms	$<$ 0.1ms
3	$\sim$ 0.2ms	$<$ 0.1ms
4	$\sim$ 0.6ms	$\sim$ 0.1ms
5	$\sim$ 3.4ms	$\sim$ 0.2ms
6	$\sim$ 27.7ms	$\sim$ 0.2ms
7	$\sim$ 314.6ms	$\sim$ 0.2ms
8	$\sim$ 5388.4ms	$\sim$ 0.3ms
9	$\sim$ 102990.9ms	$\sim$ 0.3ms
10	`Interrupted`	$\sim$ 0.4ms
...	...	...
1000	`Interrupted`	$\sim$ 1424.6ms
...	...	...
2000	`Interrupted`	$\sim$ 5683.6ms

As before, the running times displayed here are simply samples and should not be considered as statically significant values. Variations, potentially very strong, are to be expected. However, the above table still clearly shows that the previously problematic solution doesn't exhibit the same extremely bad behaviour anymore with this simple optimisation in place.

5.2 Caching of interactions

One other optimisation we can perform is caching the interactions returned from the semantics function. Indeed, since the semantic function is *pure*[1], it will always return the same list of interactions given the same connector and the same waiting list. Since we know this computation can be very expensive, we might want to cache its results. If a caching policy was in place, we could, for certain system states, instead of executing the possibly expensive semantic function, look into a cache for the list of interactions. Even though this particular optimisation was not implemented in the current version of the two frameworks, we still wish to briefly discuss its feasibility here.

Probability of cache hit

The first question that must be asked when considering caching is whether or not the function whose results should cached is frequently called with the same arguments. In this settings, the only parameter that changes is the waiting list of a system. In the presence of very cyclic systems, the waiting list might frequently be the same. This might not always be the case, especially in very dynamic systems.

Waiting list equality

In order to decide whether a certain system state, that is waiting list, has been seen, we must also understand when two waiting lists are the same. Since values contained in the waiting list might be of any given type, we can not guarantee that computing whether two such values are equal is fast, or even terminates! For instance, if the waiting list contained functions, then equality of such waiting lists is undecidable.

As using equality is impossible, we must instead rely on provable equality, a notion that we will expose here. Intuitively, we want a very fast way to know if two values are the same. To do so, we accept to be given a wrong *No* answer. However, when we are given a *Yes* answer, we are certain that the values are indeed equal.

More formally, if two values are provably equal, then they must also be equal. If they are not provably equal, then they might or might not be equal. Formulated otherwise, provable equality is equality with false negatives. This notion is sufficient to be used in our caching strategy. Indeed, since the cache is only an optimisation, it is never wrong no to use it, therefore false negatives are acceptable. However, false positives should never happen, as otherwise the value obtained from the cache might be different from the value to be computed.

[1]Purity of the semantics function is guaranteed in Haskell, since all functions can not to have any side effects by default. In Scala, this is not always true, since some functions used as part of connectors may perform unwanted side effects. However, it is generally considered bad practice to have impure functions used in places where pure functions are expected, and users of the framework could be responsible to ensure that they do not perform unwanted side effects as part of the functions they provide to connectors.

Provable equality of waiting lists

To decide whether two waiting lists are provably equal, we could compare them by the bindings from atoms-ports to values they contain. If for any two corresponding bindings the two values are not provably equal, then the waiting lists themselves are not provably equal. If a binding in one waiting list is not present in the other, then the two waiting lists are also not provably equal. Otherwise, we would consider the two waiting lists to be provably equal.

Provable equality in Scala

In Scala, the notion of provable equality of values can be transposed to the notion of referential equality, also called physical equality. Indeed, if two objects have the same references, then they by definition must be equal.

Provable equality in Haskell

In Haskell, there is no notion of reference or physical equality, since it would break referential transparency. If caching were to be implemented, we could instead declare a class `DefiniteEq` to represent types which support provable equality. We would then require the types of all values sent through ports to be instances of `DefiniteEq`.

```
class DefiniteEq a where
    (===) :: a -> a -> Bool
```

Then, standard types would be given `DefiniteEq` instances:

```
instance DefiniteEq Int where
    -- Equality is fast on ints, we can use it here.
    x === y = x == y

instance DefiniteEq [a] where
    -- Empty lists are provably equal.
    [] === [] = True
    -- Other than that, we have no way of knowing
    _  === _  = False

instance DefiniteEq (a -> b) where
    -- Two functions are never provably equal.
    f === g = False

instance DefiniteEq () where
    -- Unit is always equal to itself.
    _ === _ = True
```

Note that a too crude definition of provable equality would lead to very poor use of the cache, which could lead to very little performance improvement, or even performance deterioration, since caching is not performed for free!

5.2.1 Multiple layers of caching

So far we have only considered the possibility of caching results of the semantics functions at the top level call of the function. However, since the function we want to cache is recursive, we might have the opportunity to cache its results at many different layers in the connector.

A heuristic function could used to determine at which places within the connectors caching should take place. The heuristics could make use, amongst others, of the following criterions:

- Number of atoms involved in the connector. If this number is too large, then the use of the cache might not be worth the effort, due to the increased difficulty of cache hits.
- Precision of the provable equality relation on the underlying sent values. For instance, if the underlying ports are used to send functions as upwards values, then there is no point in trying to cache the interactions at this point, since we will never consider two waiting lists to be provably equal.
- Cost of computation. It might be worth implementing caching at places where we know that the evaluation of the semantics function on the underlying connector is expensive. Caching trivially obtained interactions might not be worth it.

  Cost could be in turn heuristically derived from the number of `OneOf` or `BothOf` combinators, since they lead to combinatorial complexity. Amongst other criterions is also the number of `Mapped` and `Guarded` combinators, since applying user functions to upwards value might be costly.

Observational provable equality on waiting lists

Using a cache at different levels within the connector also offers an opportunity to relax our definition of provably equality between two waiting lists. Indeed, the presence or not of some atoms in the waiting list plays no role over the set of interactions returned in case the connector does not involve the said atoms either via an explicit `Bind` or via `Dynamic`.

For caching purposes, we could then define two waiting lists to be observationally provably equal if and only if they are provably equal once they have been stripped of all irrelevant bindings. A binding would be considered irrelevant if the atom-pair could not be produced by an underlying explicit `Bind` or `Dynamic`.

5.2.2 Final words on caching

As we have seen, caching is a complex issue. We have explored ways to implement caching within the framework and discussed some paths that could be explored to make caching as efficient as possible. In the scope of this thesis however, we will not go any further. Deciding at which level within the connectors caching should take place, as well as cache sizes and eviction strategies, could be done as future work.

5.3 Connector optimisation

The next series of optimisations we investigate will focus on the structure of connectors. As we have already established during the formalisation of connector combinators, many connectors are equivalent. Even though they might have a different structure, they lead to the exact same set of interactions. In this section, we will use this property to bring performance improvements by performing some optimisation passes on connectors before, and even while, they are used as part of a running system. We will try to detect some inefficiencies in the structure of connectors and simplify them. In this category, we will discuss the removal of dead `Bind`s and the propagation of `Success` and `Failure`.

5.3.1 Dead `Bind` elimination

The first optimisation we can perform is pretty straightforward. During the runtime of the engine, when an atom terminates, then we know for sure that it will never be part of any new interaction again. Therefore, we can change all the `Bind` connector that involve the atom into `Failure` combinators. This optimisation can seem trivial, but it can open the way to more optimisations on the structure of the connector. It can easily be defined as:

```
-- | Eliminates from the combinator all binds that are considered dead.
deadBindElimination :: Set (AtomId s) -> Connector s d u -> Connector s d u
deadBindElimination as c = case c of
    -- Eliminates the binding in case the atom is declared dead.
    Bind a p -> if Set.member a as then Failure else c

    -- In other cases, simply recursively call all branches.
    OneOf c1 c2 -> OneOf (deadBindElimination as c1)
                         (deadBindElimination as c2)
    BothOf c1 c2 -> BothOf (deadBindElimination as c1)
                           (deadBindElimination as c2)
    Mapped f c1 -> Mapped f $ deadBindElimination as c1
    ContraMapped f c1 -> ContraMapped f $ deadBindElimination as c1
    Guarded f c1 -> Guarded f $ deadBindElimination as c1
    Feedback c1 -> Feedback $ deadBindElimination as c1
    FirstOf c1 c2 -> FirstOf (deadBindElimination as c1)
                             (deadBindElimination as c2)
    Maximal f c1 -> Maximal f $ deadBindElimination as c1
    Joined c1 -> Joined $ deadBindElimination as c1

    -- For all other leaf cases (Success, Failure, Dynamic), we do nothing.
    _ -> c
```

As can be seen, this function is very simply. In most cases, the function is simply recursively applied to children connectors. It is only in the case of `Bind` that actual work is performed, by changing the `Bind` to `Failure` when the related atom is declared dead.

5.3.2 `Failure` and `Success` propagation

Another simple optimisation we can perform is changing obviously always successful or obviously always failing connectors to `Success` and `Failure` combinators.

To do so, we must be cautious not to evaluate any function that might have been provided by the user. We do not want to execute any expensive, or even non-terminating, computation to optimise the connector. The below implementation in Haskell shows how this optimisation could be straightforwardly implemented.

```haskell
-- | Propagates failure and success up the connector.
simplify :: Connector s d u -> Connector s d u
simplify c = case c of

    -- In binary combinator cases.
    OneOf c1 c2 -> case (simplify c1, simplify c2) of
        (Failure, c2') -> c2'
        (c1', Failure) -> c1'
        (c1', c2') -> OneOf c1' c2'
    BothOf c1 c2 -> case (simplify c1, simplify c2) of
        (Failure, _) -> Failure
        (_, Failure) -> Failure
        (Success x1, Success x2) -> Success (x1, x2)
        (c1', c2') -> BothOf c1' c2'
    FirstOf c1 c2 -> case (simplify c1, simplify c2) of
        (Success x, _) -> Success x
        (Failure, c2') -> c2'
        (c1', Failure) -> c1'
        (c1', c2') -> FirstOf c1' c2'

    -- In unary combinator cases.
    Mapped f c1 -> case simplify c1 of
        Failure -> Failure
        Success x -> Success $ f x  -- f x is not forced here.
        c1' -> Mapped f c1'
    ContraMapped f c1 -> case simplify c1 of
        Failure -> Failure
        Success x -> Success x
        c1' -> ContraMapped f c1'
    Guarded f c1 -> case simplify c1 of
```

```
        Failure -> Failure
        -- Note that we do not want to evaluate the condition
        -- in here in the case of success. This might not terminate,
        -- and we can not return the connector lazily.
        c1' -> Guarded f c1'
    Feedback c1 -> case simplify c1 of
        Failure -> Failure
        Success x -> Success x
        c1' -> Feedback c1'
    Maximal f c1 -> case simplify c1 of
        Failure -> Failure
        Success x -> Success x
        c1' -> Maximal f c1'
    Joined c1 -> case simplify c1 of
        Failure -> Failure
        -- We might be tempted here to handle the Success case
        -- differently. However, we must refrain from doing so
        -- because evaluating the value contained in success
        -- might be expensive, or even not terminate.
        c1' -> Joined c1'

    -- In all other cases (Success, Failure, Bind, Dynamic), we do nothing.
    _ -> c
```

This optimisation pass could be performed every time an atom is declared dead, just after the dead `Bind` elimination pass. This would mean that with every dead atom, the connector could potentially shrink in size and complexity.

5.3.3 Engine changes

The two optimisations we have discussed require some change in the engine to be applicable. Indeed, they require the connector of a system to change while the system is running. The changes to bring to the engine to render those two optimisations possible are minimal:

- First of all, the connector should be stored in a mutable variable, such as a `MVar` in Haskell or a `var` in Scala, which should be protected against concurrent accesses.
- Secondly, and lastly, at the end of the execution of an atom, the thread responsible of the execution of the atom at that time should perform the dead `Bind` elimination phase followed by a `Failure` and `Success` propagation phase.

5.4 Early execution of stable interactions

We have seen throughout this chapter various optimisations that aim at speeding up the computation of the semantic function. In last section of this chapter we explore a different way to have more efficient systems. The way we proceed is by altering the execution model of BIP. As we have seen, BIP systems execute in a lock-step manner: All live atoms must be waiting in order for an interaction to be executed. In this setting, the time between two interactions is lower-bounded by the execution time of the slower of the atoms. This mode of execution can leave many atoms waiting for an interaction that could already be performed. In this section, we investigate the possibility of executing this type of interactions before all atoms reach a stable state.

Note that some of the ideas in this section are closely related to some developed as part of the study of the execution of BIP systems in distributed settings[27][28][29][30]. Even though we did not base any of the following optimisation on their research, we feel compelled to mention their work here.

State maximality and interaction stability

During the formalisation of the theory of connector combinators, we discussed of the notion of maximality of states and stability of interactions. We defined a state to be maximal for a given connector if enabling new ports for any given atom can not enable any new interaction. We also defined an interaction to be stable for a given state and connector if the interaction would always be proposed regardless how many new ports are activated. Those two notions will play a crucial role here. Indeed, our goal is to enable the engine to execute stable interactions before the system is in a stable state, and fall back to the regular semantics if the system reaches a stable state.

Justification

Before we proceed any further, this change of mode of execution must be justified. In some sense, the new way of executing systems that is proposed must still satisfy all the safety properties guaranteed by the previous execution model. For instance, if we are sure that no deadlock can arise using the lock-step mode of execution, we still would like to have the same guarantees using the new mode of execution.

Intuitively, this can be justified by taking the abstract view of systems as state machines that is commonly adopted in BIP. Indeed, a system in a non-stable state can simply be considered as in a certain control state, but with some outgoing transitions, in this case interactions, possibly unavailable yet. This view is valid, since we only offer *stable interactions* as transitions. Therefore, all transitions proposed in the unstable system are transitions proposed in the stable system. We say that the unstable system *simulates*[31] the stable one.

Remark. This abstract view however has its limits since atoms are no longer simple state machines. Since atoms can make side effecting computations, the new mode of execution is no longer strictly equivalent to the previous one. This is to be expected!

For instance, as the implementation does not enforce the full isolation of atoms, atoms may communicate directly or indirectly with each other by other means than interactions or even communicate with the underlying platform. By communicating, they can detect if the system is running in a lock-step fashion or not and act differently in each case. For this reason, we allow the users of the framework to specify whether or not early execution of interactions should be enabled.

Modified semantic function

In order to obtain stable interactions, we defined a modified semantic function, which, in addition to returning only stable interactions, will also indicate if the state is maximal for the given connector. This function, whose implementation has been guided by the derivations rules we have shown for maximality and stability, is implemented in Haskell as follows. Note that the following function has been optimised as so to avoid creating downwards incompatible interactions, as was previously shown earlier in this chapter. This may render the function somewhat difficult to comprehend!

```
-- | Returns for the given connector and given waiting list
--   a list of provably stable interactions that do not use any
--   of the specified atoms.
--
--   Also indicates if the list of interactions is exhaustive,
--   that is if the state is maximal.
getStableInteractionsWithout :: Set (AtomId s)
                              -> Connector s d u
                              -> WaitingList s
                              -> ([OpenInteraction s d u], Bool)
getStableInteractionsWithout as c w = case c of
    -- Core combinators
    Success x    -> ([OpenInteraction x (const []) (Set.empty)], True)
    Failure      -> ([], True)
    Bind a p     -> case getBoundInteraction a p w of
        Nothing -> ([], False)
        Just o  -> (if Set.member a as then [] else [o], True)
    OneOf c1 c2  ->
        let (os1, m1) = getStableInteractionsWithout as c1 w
            (os2, m2) = getStableInteractionsWithout as c2 w
        in (os1 ++ os2, m1 && m2)
    BothOf c1 c2 ->
        let (os1, m1) = getStableInteractionsWithout as c1 w
            (oss2, m2s) = unzip [ getStableInteractionsWithout
                                     (Set.union as $ getInvolved o1) c2 w
                                | o1 <- os1 ]
        in (, m1 && and m2s) $ do
```

```
            -- The state is maximal for BothOf if it is maximal
            -- for c1 and maximal for c2.
            (o1, os2) <- zip os1 oss2  -- Zips together the open interactions
                                       -- of c1 with the corresponding open
                                       -- interactions of c2.
            o2 <- os2
            let o3 = OpenInteraction
                    { getUpwards   = (getUpwards o1, getUpwards o2)
                    , getDownwards =
                        \ x -> getDownwards o1 x ++ getDownwards o2 x
                    , getInvolved  =
                        Set.union (getInvolved o1) (getInvolved o2)
                    }
            return o3

    -- Data combinators
    Mapped f c1 ->
        let (os1, m1) = getStableInteractionsWithout as c1 w
        in (, m1) $ do
            o1 <- os1
            return $ o1 { getUpwards = f (getUpwards o1) }
    ContraMapped f c1 ->
        let (os1, m1) = getStableInteractionsWithout as c1 w
        in (, m1) $ do
            o1 <- os1
            return $ o1 { getDownwards = getDownwards o1 . f }
    Guarded f c1 ->
        let (os1, m1) = getStableInteractionsWithout as c1 w
        in (, m1) $ do
            o1 <- os1
            guard (f $ getUpwards o1)
            return o1
    Feedback c1 ->
        let (os1, m1) = getStableInteractionsWithout as c1 w
        in (, m1) $ do
            o1 <- os1
            return $ o1 { getDownwards =
                \ x -> getDownwards o1 (x, getUpwards o1) }

    -- Priority combinators
    FirstOf c1 c2 -> case getStableInteractionsWithout Set.empty c1 w of
        -- No stable interaction, and no possibility to
        -- have such an interaction in the future.
        -- Therefore we respect the priority enabling c2.
```

```
        ([], True) -> getStableInteractionsWithout as c2 w
        -- There are, or they might be in the future, interactions
        -- coming from c1, therefore we propagate them.
        (os1, m1) ->
            (mfilter (not . any (flip Set.member as) . getInvolved) os1, m1)
    Maximal f c1 ->
        let (os1, m1) = getStableInteractionsWithout Set.empty c1 w
        in if m1 then
            -- The state is maximal. Therefore there won't
            -- possibly be any interaction in the future with
            -- higher priority.
            let upwardsValues = fmap getUpwards os1
                biggerThan x = \ y -> f y x == GT
                isMaximal x = not $ any (biggerThan x) upwardsValues
            in (mfilter (not . any (flip Set.member as) . getInvolved)
               $ mfilter (isMaximal . getUpwards) os1, True)
        else
            -- Their might be interactions with higher priority
            -- in the future, thus there is no (provably) stable
            -- interactions at this point.
            ([], False)

    -- Dynamic combinators
    Dynamic p -> (mfilter (not . any (flip Set.member as) . getInvolved) $
                    getBoundInteractions p w, False)
    Joined c1 ->
        let (os1, m1) = getStableInteractionsWithout as c1 w
            (oss2, m2s) = unzip [ getStableInteractionsWithout
                                    (Set.union as $ getInvolved o1)
                                    (getUpwards o1) w
                                | o1 <- os1 ]
        in (, m1 && and m2s) $ do
            -- The state is maximal for Joined if it is maximal
            -- for c1 and maximal for all upwards values, which are
            -- themselves connectors.
            (o1, os2) <- zip os1 oss2  -- Zips together the open interactions
                                       -- of c1 with the open interactions
                                       -- of the connector in
                                       -- its upwards value.
            o2 <- os2
            let o3 = OpenInteraction
                    { getUpwards   = getUpwards o2
                    , getDownwards =
                        \ x -> getDownwards o1 x ++ getDownwards o2 x
```

```
                        , getInvolved  =
                            Set.union (getInvolved o1) (getInvolved o2)
                        }
            return o3
```

From this function, we derive the two following functions that will be used within the engine.

```
-- | Returns for the given connector and given waiting list
--   a list of provably stable interactions.
getStableClosedInteractions :: ClosedConnector s -> WaitingList s
                            -> [Interaction s]
getStableClosedInteractions (ClosedConnector c) w =
    fmap close $ getStableInteractions c w

-- | Returns for the given connector and given waiting list
--   a list of provably stable interactions.
getStableInteractions :: Connector s d u -> WaitingList s
                      -> [OpenInteraction s d u]
getStableInteractions c w = fst $ getStableInteractionsWithout Set.empty c w
```

Modifications to the engine

Now that we have defined this stable semantics functions, we must adapt the engine to use it make use of it. The changes to perform are minimal. We must make the following changes to worker threads and to the main loop.

Worker thread changes

The main loop, which executes interactions and spawn atom continuations, must be signalled each time a worker thread terminates its job, that is every time an atom terminates or waits. This will ensure that the main loop may be given a chance to execute interactions before the system is in a stable state. A change must therefore be made to the function called at the end of worker threads. Instead of its previously seen implementation, the `spawnAction` function must start like shown here:

```
-- | Spawns an atom on its own lightweight thread.
--
--   The number of threads MUST be adapted accordingly before calls
--   to this function are made.
spawnAtom :: SystemOptions -> SystemState s -> Task s -> IO ()
spawnAtom opts state (Task a x) =
    void $ flip forkFinally terminateThread $ interpret $ getInstructions x
  where
    terminateThread _ = do
```

```
        -- We decrement the number of running atoms.
        n' <- decrementAndFetch (getRunningVar state)

        -- If there isn't any atom left running,
        -- or if early evaluation of stable interactions is enabled,
        -- we notify the engine.
        when (n' == 0 || getEarlyInteractionExecution opts) $
            signal (getSemaphore state)
```

The important part is in the condition used to check whether the main loop must be awoken or not. Before, we required that no atoms are still running. Now, we also signal if early execution of stable interactions is enabled, regardless how many atoms here still executing.

Main loop changes

Most of the changes to the engine are made to its main loop. The main loop, after been signalled, must check whether it is in a stable state and choose the semantics function accordingly. If the system is stable, the standard semantic function should be used. In case the system is not yet stable, only stable interactions must be obtained, and thus the newly defined semantic function is used instead. Once this is done, we look at the possible interactions.

- If no non-empty interactions are possible, we must act differently whether the system is stable or not.
  - In case the system is stable, then we know for sure that no new interactions will later be enabled. The function can thus terminate
  - On the other hand, if the system was not stable, then we will loop back and wait for a signal to continue from one of the remaining atoms.
- If some interactions are possible, we pick one and update the waiting list accordingly. Before we loop back and spawn the continuations of atoms however, we signal ourself. Indeed, there might be already stable interactions ready to be executed!

The implementation of the main loop is shown below.

```
-- | Main loop of the engine.
engineLoop :: SystemOptions -> ClosedConnector s -> SystemState s
           -> [Task s] -> IO ()
engineLoop opts c state as = do

    when (not $ null as) $ do
        --  Updates the number of running atoms.
        incrementBy (getRunningVar state) (toInteger (length as))
```

```
        -- Spawning all continuations of atoms.
        forM_ as (spawnAtom opts state)

    -- Waiting on the semaphore to be notified.
    -- This means that we can now try to execute the next interaction.
    wait (getSemaphore state)

    -- Reading the current state of the waiting list.
    w <- takeMVar (getWaitingListVar state)

    -- Snapshot of the number of running threads.
    -- If n is 0, then we know for sure that there
    -- aren't any atoms left running.
    -- If n is larger,
    -- we are garranteed to be signaled at least once more.
    n <- readMVar (getRunningVar state)

    let stable = n == 0

    -- Picks the semantic depending on whether the system is stable or not.
    let semantic = if stable
        then getClosedInteractions
        else getStableClosedInteractions

    -- Trying to get an interaction
    case filter (not . null) $ semantic c w of
        [] -> if stable
            then do
                return ()    -- No non-empty interactions to perform.
                             -- Might be a deadlock, or not, depending on
                             -- the state of the waiting list.
            else do
                -- We loop back, since the system wasn't yet in a stable state.
                -- We do not schedule any new tasks however.
                -- Before we do so, we restore the waiting list.
                putMVar (getWaitingListVar state) w
                engineLoop opts c state []

        is -> do
            -- We have possibly multiple interactions.
            -- We let the interaction picker decide.
            i <- getInteractionPicker opts is
```

```
-- Update the waiting list.
-- All atoms of the interactions are no longer waiting.
putMVar (getWaitingListVar state) $
    foldl' (flip setRunning) w $ fmap getAtom i

-- There might be another possible interaction,
-- we signal ourself to try to get an interaction yet again on next
-- iteration of the loop.
when (getEarlyInteractionExecution opts) $
    signal (getSemaphore state)

-- We loop using the given interaction as our list
-- of continuations to execute.
engineLoop opts c state i
```

With this change in place, stable interactions can now be run even though the system is not yet in a stable state. This can lead to arbitrarily large performance improvements in some applications, as atoms no longer must all wait for the slowest of them. We will evaluate the performance impact of this optimisation while looking at some example systems in the next chapter.

Conclusion

We conclude this chapter with this last optimisation. Throughout this chapter, we have seen some tricks and changes that can lead to drastic improvements in some use cases. In the next chapter, we will actually implement concurrent applications using the Scala and Haskell BIP frameworks. This exposition will show the expressivity of the framework as well as give us a chance to actually measure the performance of the engine in typical settings.

Chapter 6

Examples and evaluation

In this section, we explore examples of applications that can be written in the developed frameworks. For the different example systems, we discuss the expressivity of the frameworks and their performance. The performance measurements were taken on a 2.8 GHz Intel Core i7 MacBookPro with 16 GB of 1600 MHz DDR3 RAM. The Haskell measurements where taken using the `criterion` benchmarking framework[37]. The programs were all compiled with compiler optimisations on[1].

6.1 Token ring (Haskell)

The first example application we see is the so-called *token ring* application, in which a token is passed between multiple components in a circular manner. For this example, we will use the Haskell framework.

This example is straightforwardly ported to the framework. Indeed, each component corresponds to an atom. Each atom can either wait on a `receive` port to receive the token, or send it through the `send` port.

The connector simply synchronises the send port of all atoms with the receive port of the next atom in the ring.

6.1.1 Implementation

```
-- | Describes a token ring.
--
--   * @n@ is the number of atoms.
--
--   * @m@ is the number of token exchanges.
--
--   * @display@ indicates if atoms should print to standard output or not.
tokenRing :: Int -> Int -> Bool -> System s ()
tokenRing n m display = mdo
```

[1] `-O3` on ghc

```
-- Port on which to send the token and its recipient.
send <- newPort

-- Port on which to receive the token.
receive <- newPort

-- Vector of atom identifiers
let v = fromList as

-- Returns the identifier of the atom that is
-- supposed to receive message from the i^th atom.
let next i | i == pred n = a
           | otherwise   = v ! i

-- Behaviour of the i^th atom.
let atom i = do
        -- Awaiting to receive a value.
        n <- await receive ()
        -- Printing the value received.
        when display $ liftIO $ do
            putStrLn $ "Atom " ++ show i ++ " received " ++ show n
        -- Computing the next value.
        let !n' = succ n
        -- Sending the value to the next atom.
        when (n' <= m) $ do
            await send (n', next i)
            atom i

-- Creating the first atom.
a <- newAtom $ do
    -- Printing the first message.
    when display $ liftIO $ putStrLn "Atom 0 ready to send."
    -- Sending the first message.
    await send (1, next 0)
    -- From now on, acts as a "normal" atom.
    atom 0

-- Creating the n - 1 next atoms.
as <- forM [ 1 .. pred n ] $ \ i ->
    newAtom $ atom i


-- Connector of the system.
```

```
registerConnector $ do
    -- Accepts any atom on the port send.
    (value, destination) <- dynamic send
    -- Bind the receiver on the receive port
    -- and send it the given value.
    bind destination receive # receiving value
```

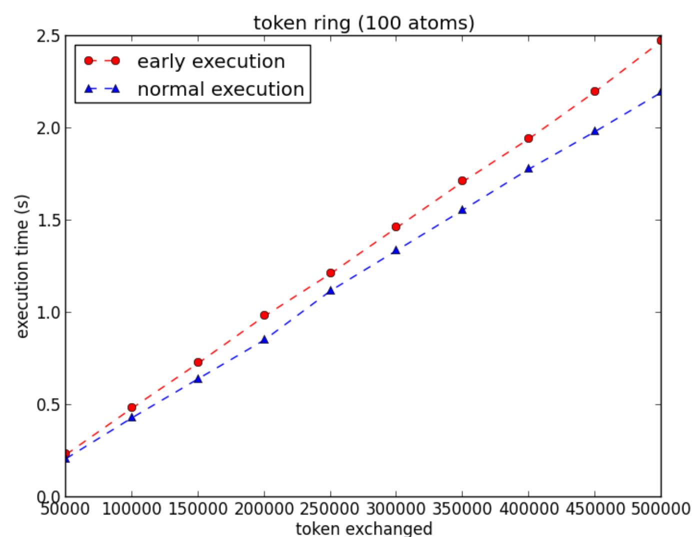
6.1.2 Expressivity

The above example illustrates very well the improvement brought by the embedding of BIP in an expressive language. The above system description makes use of loops to create the atoms, each of which with a specific behaviour.

However, the biggest expressivity gain is to be found in the declaration of the connector. Using the `do`-notation and dynamic combinators, we have been able to concisely and precisely describe the possible interactions of the system. The connector is arguably very simple to write and, perhaps more importantly, to understand.

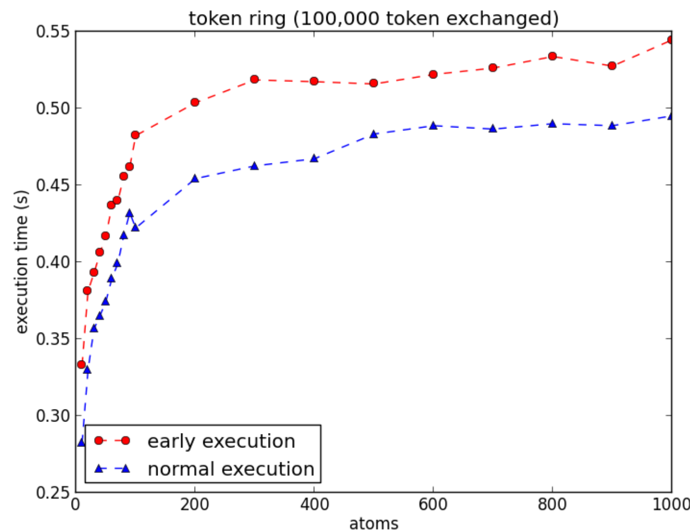
6.1.3 Performance

To evaluate the performance of the Haskell framework on this specific example, we performed two benchmarks. The first looks at the running time for a fixed set of atoms and a varying number of token exchanges. The second shows the impact of an increase of the number of atoms for a fixed number of token exchanges. Those benchmarks are also the occasion to show the impact of early evaluation of interactions in a situation when it can not be used.



As can be clearly seen from the first benchmark, an increase in the number of token exchanges leads to a linear increase in execution time, as would be expected. We also see that the cost of enabling early execution of stable interactions in this case leads

to a slight decrease in performance. This is expected since the system offers little to no possibility of executing stable interactions earlier. The system indeed almost immediately reaches a stable state since atoms perform no actual work.



In the second benchmark, where the number of token exchanges was fixed, we can clearly see the impact an increase of the number of atoms has on the performance of the framework. In this particular case, we can clearly distinguish a *log* shaped curve. The performance decreases logarithmically with the number of atoms live in the system. This behaviour is explained by the logarithmic complexity of operations on the waiting list. Indeed, the current implementation of the waiting lists makes heavy use the standard `Map` data type of the Haskell library, which is implemented as a balanced tree.

As with before, the performance when early execution of stable interactions is enabled is slightly worse, for exactly the same reasons we have previously outlined.

6.2 Masters-Slaves (Scala)

The next example models a system composed of master and slave components. The behaviour of the slaves is very simple. They can either be free, or be reserved by a master. The slaves simply continuously loop between those two states.

The behaviour of the masters is a bit more involved. Each master first sequentially requests two slaves, and then uses the two slaves in some computation. Once this is done, the master loops back to its initial state.

In the below implementation in Scala, masters and slaves are implemented as individual atoms. The connector of the system simply specifies that any number of masters may reserve a slave or use their two previously reserved slaves.

6.2.1 Implementation

```
/** Dining philosophers example */
object MastersSlaves {

  /** Starts the example. */
  def main(args: Array[String]) {

    // Number of masters.
    val m = 30

    // Number of slaves
    val s = 60

    // Creating the system.
    val system = new System

    // Disables early execution of stable interactions.
    system.setEarlyExecution(false)

    // Ports of slaves.
    val reserve = system.newPort[Any, Atom]
    val use     = system.newPort[Any, Unit]

    // Creation of the n slaves.
    val slaves: Array[Atom] = for {
      i <- (0 until s).toArray
    } yield system.newAtom {

      // Defines the behaviour of the slave.
      def act() {
        // Sending the identifier of the atom to the port.
```

```
      await(reserve, getSelf()) { (_: Any) =>
        println("Slave " + i + " now reserved")
        // The slave is now reserved.
        await(use) { (_: Any) =>
          // The slave has been used and can now
          // loop back to its initial state.
          act()
        }
      }
    }

    // Starting the behaviour.
    act()
  }

  // Ports of masters.
  val request = system.newPort[Atom, Unit]
  val compute = system.newPort[Any, (Atom, Atom)]

  // Creation of the m masters.
  val masters: Array[Atom] = for {
    i <- (0 until m).toArray
  } yield system.newAtom {
    var j = 1;

    def act() {
      // First, request a first atom.
      await(request) { (first: Atom) =>
        // Then, request a second atom.
        await(request) { (second: Atom) =>
          // Triggers a computations,
          // using the two previously requested atoms.
          await(compute, (first, second)) { (_: Any) =>
            println("Master " + i + " computing for the time " + j)
            j += 1
            // The master loops back to its original state.
            act()
          }
        }
      }
    }

    // Starting the behaviour.
    act()
```

```
    }

    // For each master, the following array contains
    // a connector that requests a slave for the connector.
    val reqs = for {
      i <- 0 until m
    } yield {
      // Synchronises the master with any of the slaves.
      close(masters(i).bind(request).andRight(dynamic(reserve)))
    }

    // For each master, the following array contains
    // a connector that uses the two previously reserved slaves.
    val comps = for {
      i <- 0 until m
    } yield {
      masters(i).bind(compute).flatMap { (atoms: (Atom, Atom)) =>
        close(atoms._1.bind(use).and(atoms._2.bind(use)))
      }
    }

    // The connector simply states that an interaction
    // consists of many requests and computations
    system.registerConnector(manyOf(reqs ++ comps : _*))

    // Starts the system.
    system.run()
  }
}
```

6.2.2 Expressivity

The above example shows most of the features of the framework. Using the combinators we have defined, the connectors have been concisely and precisely described. The high level constructs proposed by the host language bring large improvements in usability and flexibility.

However, we feel that having to resort to a recursive function instead of a normal loop in the behaviour of atoms is a bit awkward. This is however unavoidable since the `await` function never returns normally. A library of functions mimicking loops could be developed to alleviate this problem, as was already done in the context of the Scala actors[32] library.

6.3 Producers-Consumers (Haskell)

Another classic concurrent system example is the producers-consumers problem, in which some components are responsible to create some values and some others are responsible to consume them.

6.3.1 Implementation

```
-- | Producers-consumers system.
--
--   * @n@ is the number of producers.
--
--   * @m@ is the number of consummers.
--
--   * @k@ is the number of values produced.
--     It should be a multiple of @n@.
--
--   * @s@ is the cost in microseconds to produce a value.
producersConsumers :: Int -> Int -> Int -> Int -> System s ()
producersConsumers n m k s = do
    -- Creation of the two ports.
    send    <- newPort  -- Port on which to send values produced.
    request <- newPort  -- Port on which to receive values.

    -- Creation of the producers.
    replicateM n $ newAtom $ replicateM (k `div` n) $ do
        when (s > 0) $ liftIO $ threadDelay s
        await send ()

    -- Creation of the consumers.
    replicateM m $ newAtom $ forever $ await request ()

    -- The connector of the system
    registerConnector $
        dynamic send  -- Picks any producer
        <*
        dynamic request  -- Picks any consumer
```

6.3.2 Expressivity

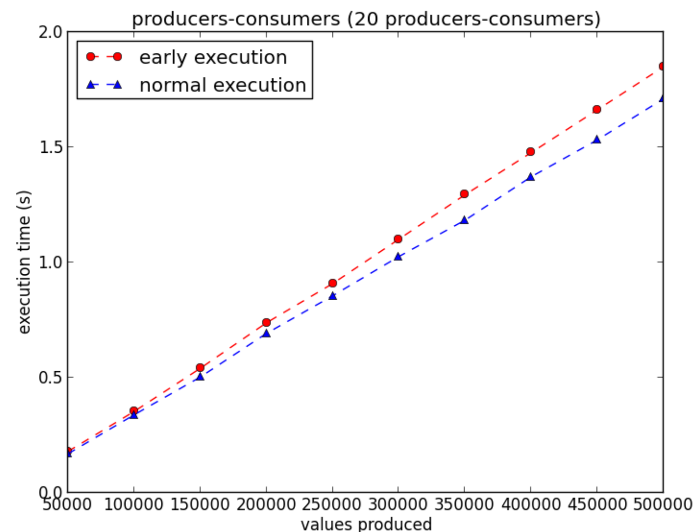
Just as in the token ring example, we were able to make use of high level constructs made available by the host language to easily and precisely describe the system.

The connector, very simply indicates that any consumer can receive values from any producer. Note that the connector has been written using an *applicative* style, in

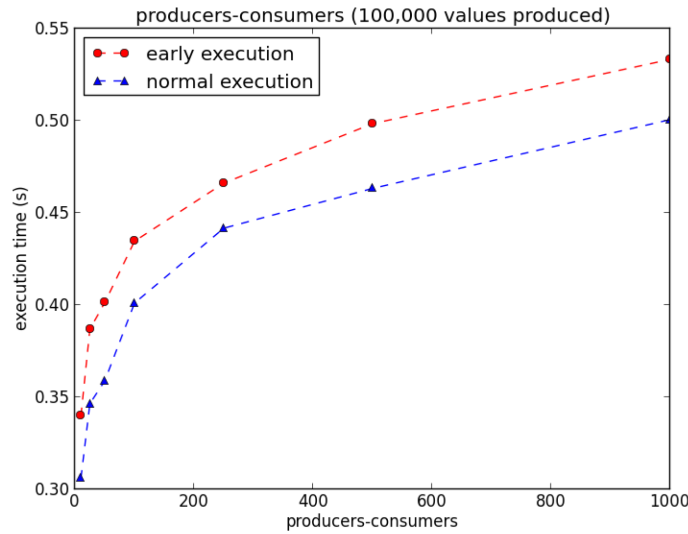
which functions from the `Applicative` type class are used. Since connectors have an instance of such a type class, users of the framework can make use of functions such as `<*` and `*>`, which they may be familiar with, to describe their connectors.

6.3.3 Performance

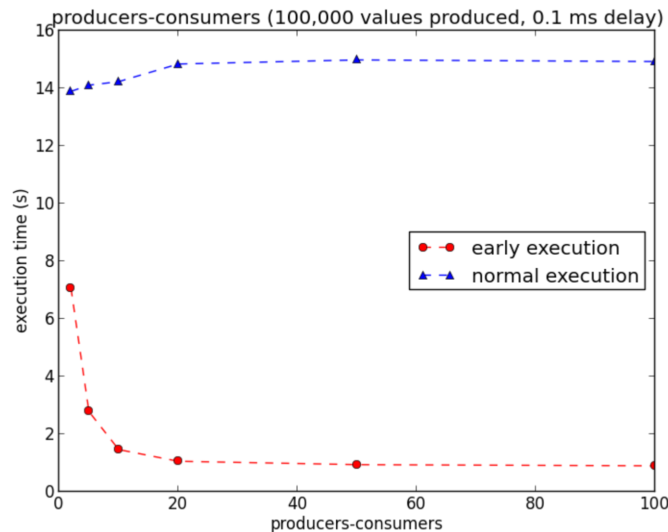
We evaluate the performance of the application on the Haskell framework with three different benchmarks. In the first two, we consider that the cost of producing a value is null, so that we have an idea of the overhead the frameworks brings. We measure how the running time of the system changes when we increase the number of value produced and the number of atoms. In the last benchmark, we investigate the case when producing a value has a non-zero cost for the producer. Note that, in all the benchmarks, we have fixed the number of producers to be equal to the number of consumers.



As can be seen in the first benchmark, the execution time of the system augments linearly with the number of values to be produced, as expected. We can also see that early evaluation of stable interactions is slightly detrimental in the case when producers do not incur any cost to produce a value.



In this benchmark, we also clearly see the previously explained logarithmic increase in running time that comes with an increase in the number of atoms of the system. In this case, since producers have no actual cost of producing a value, increasing the number of atoms is detrimental to the overall performance of the system.



For the last benchmark, we specified that producing a value actually takes some time (at least 0.1ms). In this particular benchmark, which should be more representative of an actual system, we can clearly see that the early evaluation of stable interactions strategy can lead to massive increase of performance. Indeed, the value of a waiting producer can be consumed even though some other producers are still busy.

6.4 Dining Philosophers (Haskell)

The next example we will see is the famous *Dining Philosophers* problem. In this problem, a certain number of philosophers are place around a table to eat and discuss philosophy! All of the philosophers have the same behaviour: They alternate between eating and discussing (or sleeping).

Between each two philosophers can be found a unique fork, which can only be used by any of the two philosophers next to it. The problem is that in order to eat, a philosopher must use the two forks situated next to him. This means that philosophers may not eat at the same time as their direct neighbours.

6.4.1 Implementation

Below is the implementation of the problem in Haskell. We will see many version of this system, each using a different connector. This will expose the different combinators and their properties.

The first part of the system defines the behaviour of forks, which are implemented as atoms, and of philosophers, also implemented as individual atoms.

We then build for each philosopher two connectors: One that synchronises eating with the taking of two forks, and another that synchronises sleeping with the release of the two forks. Those connectors are communicated to the rest of the system, which will define the connector to use.

```
-- | Dining philosophers problem.
--
--   * @n@ is the number of philosophers.
--
--   * @k@ is the number of meals per philosopher.
--
--   * @eat@ is the action to execute when a philosopher eats.
--
--   * @sleep@ is the action to execute when a philosopher sleeps.
diningPhilosophers :: Int -> Int
                   -> (Int -> Int -> Action s ())
                   -> (Int -> Int -> Action s ())
                   -> System s ([Connector s d1 Int], [Connector s d2 ()])
diningPhilosophers n k eat sleep = do

    allocateFork <- newPort  -- Indicates the allocation of a fork
    releaseFork  <- newPort  -- Indicates the release of a fork

    -- Creating the n forks
    forks <- replicateM n $ newAtom $ forever $ do
        -- the fork is free
```

```
    await allocateFork ()
    -- the fork is now taken
    await releaseFork ()

startEating <- newPort  -- Indicates a desire to start eating
endEating   <- newPort  -- Indicates a desire to stop eating

-- Creating the n philosophers
philosophers <- fmap fromList $ forM [1 .. n] $ \ i ->
    newAtom $ forM_ [1 .. k] $ \ j -> do
        -- the philosopher is hungry
        await startEating j  -- j indicates how many times we've eaten
        -- the philosopher eats
        eat i j
        -- the philosopher has finished eating
        await endEating ()
        -- the philosopher sleeps
        sleep i j


-- Indices of left and right forks
let right i = (succ i) `rem` n
    left i = i

-- For philosopher i to start eating,
-- the following must synchronise.
let enterEating i =
        bind (forks ! left i) allocateFork
        *>
        bind (forks ! right i) allocateFork
        *>
        bind (philosophers ! i) startEating

-- For philosopher i to stop eating,
-- the following must synchronise.
let leaveEating i =
        bind (forks ! left i) releaseFork
        <>
        bind (forks ! right i) releaseFork
        <>
        bind (philosophers ! i) endEating

-- Returning the connectors.
return ([enterEating i | i <- [0 .. pred n]]
```

```
          ,[leaveEating i | i <- [0 .. pred n]])
```

As explained above, the system does not register a connector. Instead, the system must be part of another system, which will define the connector. Below are the different version of this example, each using a different connector.

Connector using anyOf

The system below completes the example by registering a connector using the `anyOf` combinator. The connector specifies that an interaction is simply a unique philosopher starting or stopping eating. This should be used with the early execution of stable interactions in order to have a concurrently running system.

```
diningPhilosophersAnyOf n k eat sleep = do
    (es, ls) <- diningPhilosophers n k eat sleep

    registerConnector $ anyOf
        [ anyOf [ sending () e | e <- es ]
        , anyOf ls ]
```

Connector using manyOf

The next system completes the example by registering a connector using the `manyOf` combinator[2]. The connector specifies that an interaction is a non-zero number of philosophers starting or stopping eating. This version doesn't require the early execution of stable interactions to have some level of concurrency.

```
diningPhilosophersManyOf n k eat sleep = do
    (es, ls) <- diningPhilosophers n k eat sleep

    registerConnector $ allOf
        [ manyOf [sending () e | e <- es]
        , manyOf ls ]
```

Connector using anyOf and priority

The problem with the above two versions is that they are not fair. Indeed, some philosophers will get to eat more quickly than others. As an example, here is the trace of the execution using the above connectors, using 3 philosophers eating 3 times each.

```
Philosopher 1 eating (1)
Philosopher 1 sleeping (1)
Philosopher 1 eating (2)
```

[2] As a reminder, the `manyOf` involves any subset of the underlying connectors

```
Philosopher 1 sleeping (2)
Philosopher 1 eating (3)
Philosopher 1 sleeping (3)
Philosopher 2 eating (1)
Philosopher 2 sleeping (1)
Philosopher 2 eating (2)
Philosopher 2 sleeping (2)
Philosopher 2 eating (3)
Philosopher 2 sleeping (3)
Philosopher 3 eating (1)
Philosopher 3 sleeping (1)
Philosopher 3 eating (2)
Philosopher 3 sleeping (2)
Philosopher 3 eating (3)
Philosopher 3 sleeping (3)
```

In order to enforce some level of fairness, we will in this case add a priority policy to the connector. The next version specifies that philosophers may only eat if they've not eaten more than any other philosophers that also wants to eat.

```
diningPhilosophersAnyOfPriority n k eat sleep = do
    (es, ls) <- diningPhilosophers n k eat sleep

    registerConnector $ anyOf
        [ sending () $ minimal $ anyOf [ e | e <- es ]
        , anyOf ls ]
```

Using this connector, the system is totally fair, as shows the following trace:

```
Philosopher 1 eating (1)
Philosopher 1 sleeping (1)
Philosopher 2 eating (1)
Philosopher 2 sleeping (1)
Philosopher 3 eating (1)
Philosopher 3 sleeping (1)
Philosopher 1 eating (2)
Philosopher 1 sleeping (2)
Philosopher 2 eating (2)
Philosopher 2 sleeping (2)
Philosopher 3 eating (2)
Philosopher 3 sleeping (2)
Philosopher 1 eating (3)
Philosopher 1 sleeping (3)
Philosopher 2 eating (3)
Philosopher 2 sleeping (3)
```

```
Philosopher 3 eating (3)
Philosopher 3 sleeping (3)
```

Unfortunately, the system can not expose any level of concurrency, even with early execution of stable interactions. Indeed, due to the priority in the system, no interaction that may enable a philosopher to eat is provably stable for the engine.

Connector `anyOf` and shuffling

Instead of using a priority, we can use a dynamic combinator that will return each time a different order of the philosophers, thereby ensuring fairness.

To do so, another atom must be created in the system. Its goal will be to continuously shuffle the list of connectors and sent that list on a given port.

```
diningPhilosophersAnyOfShuffle n k eat sleep = do
    (es, ls) <- diningPhilosophers n k eat sleep

    shuffled <- newPort
    shuffler <- newAtom $ forever $ do
        es' <- liftIO $ shuffleM [close e | e <- es]
        await shuffled es'

    let entering = do
            es' <- bind shuffler shuffled
            anyOf es'

    let leaving = anyOf ls

    registerConnector $ anyOf
        [ entering
        , leaving ]
```

As the connector may only return interactions that involve a single philosopher, the system will not display any level of concurrency unless, as always, early execution of stable interactions is enabled.

Connector `manyOf` and shuffling

The final version we will investigate has all the properties we may desire. It shows a high level of concurrency even in the case when early execution of stable interactions is disabled, and ensures that philosophers are treated in a fair way.

```
diningPhilosophersManyOfShuffle n k eat sleep = do
    (es, ls) <- diningPhilosophers n k eat sleep

    shuffled <- newPort
```

```
shuffler <- newAtom $ forever $ do
    es' <- liftIO $ shuffleM [close e | e <- es]
    await shuffled es'

let entering = do
        es' <- bind shuffler shuffled
        ensuring (not . null) $ manyOf es'

let leaving = manyOf ls

registerConnector $ manyOf
    [ entering
    , leaving ]
```

6.4.2 Expressivity

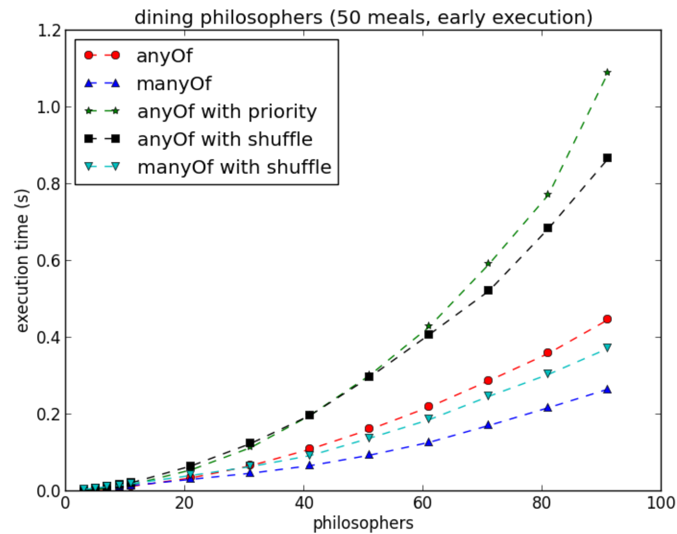
As we have seen, we have been able to express in several concise ways the coordination between the various philosophers. Avoiding deadlocks, which is non trivial in many other paradigms, is done by construction.

However, we have seen that ensuring fairness was not straightforwardly achieved. Using a priority, which seemed like a natural way to go, wasn't entirely satisfactory. The version with priority is, as we will see, not really efficient. This is due to the fact that early execution of stable interactions can not perform really well in the case of systems with priorities. In addition, using `Maximal`, which forces the entire stream of interactions to be evaluated, is generally not very efficient.

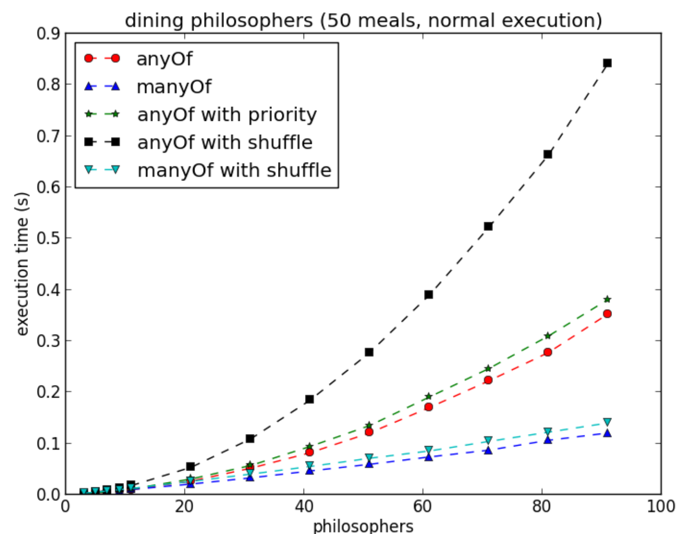
Related to this last point, you may have noticed that the `manyOf` version with priority has not been showcased. The reason is that it would be terribly inefficient. Indeed, the `Maximal` combinator forces the engine to generate all underlying interactions, which might be really costly. As `manyOf` produces an exponential number of interactions, applying a `Maximal` combinator on top of it will be exponentially inefficient.

6.4.3 Performance

Below is shown the performance of the different versions we have exposed. In the first benchmark, we show the bare cost of the framework, by having no actions performed by the philosophers when they eat and when they sleep. In the first chart, we display this benchmark with early execution of stable interactions enabled. Note that we only considered odd numbers of philosophers, since otherwise there is only little contention over the forks.

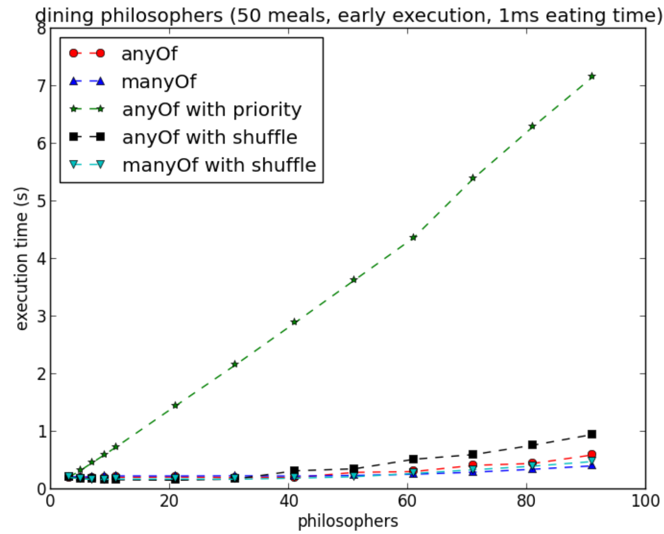


The slowest of the version in this case is the `anyOf` version with priority. This is due to the fact that this version, no two philosophers can eat at the same time. The next benchmark shows the same results for when early execution of stable interactions is disabled.

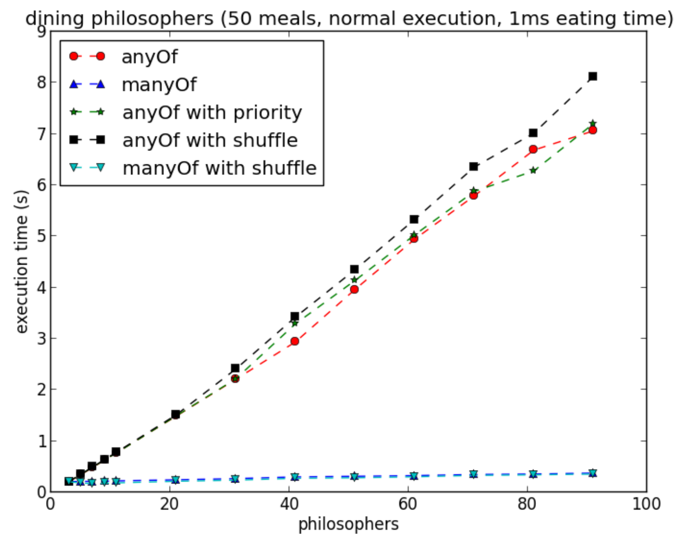


As we can clearly see, the `anyOf` versions are slower than the `manyOf` versions. This is due to the fact that the latter can lead to many philosophers eating at the same time.

In the previous benchmarks, we specified that philosophers spent no time at all eating or sleeping. In the next two benchmarks, we specify that eating takes at least 1ms. The first benchmark has early execution of stable interactions enabled.

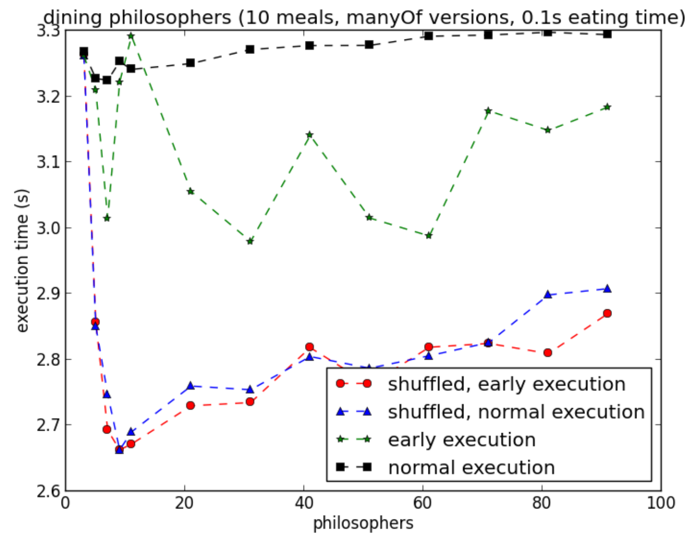


The second version, which has early execution of stable interactions disabled, is shown below:



As can be clearly seen from the above two benchmarks, the `manyOf` versions are faster than any other versions, without or without early execution of stable interactions enabled. We also see that the shuffled version, which is fair to the philosophers, has roughly the same performance as the unfair version.

A final observation we make is that the `manyOf` versions scale really well with the number of philosophers. Indeed, the performance is almost the same regardless of the number of philosophers, which indicates that the systems are able to run very concurrently. This fact is also shown by the following chart, which shows the execution time for a system of an odd number of philosophers, each of which eats 10 meals, each meal taking at least 100ms.



We can clearly see that the shuffled version performs well, regardless of whether or not early execution of stable interactions is enabled or not. The fact that the shuffled versions performed better is explained by the fact that they are fair. In the normal versions, as the number of philosophers is odd, one of the philosophers will get to start eating only once all others have finished, thereby slowing down the entire system.

6.5 Dining Philosophers (Scala)

In this section, we will revisit again the Dining Philosophers example, this time implemented in Scala.

The implementation follows the Haskell implementation very closely. In the first part of the system, forks and philosophers are declared as atoms. Then, for each philosopher, two connectors are created. The first connector synchronises eating with the taking of the two forks, and the second sleeping with the release of the two forks. As in the Haskell version, a distinct atom is created to shuffle the above connectors to ensure fairness. The connector of the system then simply requests a shuffled list of connectors and applies the `manyOf` combinator on it, which specifies that any subset of the connectors may get involved.

6.5.1 Implementation

```
/** Dining philosophers example */
object DiningPhilosophers {

  /** Starts the example. */
  def main(args: Array[String]) {

    // Number of philosophers.
    val n = 21

    // Creating the system.
    val system = new System

    // Ports of forks.
    val allocateFork = system.newPort[Any, Unit]
    val releaseFork  = system.newPort[Any, Unit]

    // Creation of the n forks.
    val forks: Array[Atom] = for {
      i <- (0 until n).toArray
    } yield system.newAtom {

      // Defines the behaviour of the fork.
      def act() {
        // The fork is free.
        await(allocateFork) { (_: Any) =>
          // The fork is taken.
          await(releaseFork) { (_: Any) =>
            act()  // The fork restarts its behaviour.
          }
```

```
      }
    }

    // Starting the behaviour.
    act()
  }

  // Ports of philosophers.
  val startEating = system.newPort[Any, Unit]
  val endEating   = system.newPort[Any, Unit]

  // Creation of the n philosophers.
  val philosophers: Array[Atom] = for {
    i <- (0 until n).toArray
  } yield system.newAtom {
    var j = 1;

    // Behaviour of the philosopher.
    def act() {

      // The philosopher waits to start eating.
      await(startEating) { (_: Any) =>
        // The philosopher can now eat.
        println("Philosopher " + i + " eating. (" + j + ")")
        j += 1
        // Eating takes some amount of time.
        //Thread.sleep(1000)

        // The philosopher now requests to start sleeping.
        await(endEating) { (_: Any) =>
          println("Philosopher " + i + " sleeping.")

          // The philosopher restarts its behaviour.
          act()
        }
      }
    }

    // Starting the behaviour.
    act()
  }

  // Contains a connector that requests both forks and
  // makes the philosopher start eating for each philosopher.
```

```
val eatings = for {
  i <- 0 until n
} yield allOf(
  forks(i) bind allocateFork,
  forks((i + 1) % n) bind allocateFork,
  philosophers(i) bind startEating)

// Contains a connector that release both forks and
// makes the philosopher start sleeping for each philosopher.
val sleepings = for {
  i <- 0 until n
} yield allOf(
  forks(i) bind releaseFork,
  forks((i + 1) % n) bind releaseFork,
  philosophers(i) bind endEating)

// Port on which the shuffled list of connectors is sent.
val shuffled = system.newPort[Any, Seq[Connector[Any, Any]]]

// Atom that continuously shuffle the list of connectors.
val shuffler = system.newAtom {

  // Behaviour of the shuffler
  def act() {
    // Shuffling the connectors.
    val connectors = Random.shuffle(eatings)
    await(shuffled, connectors) { (_: Any) =>
      act()  // Looping.
    }
  }

  // Starting the behaviour.
  act()
}

// Registering the connector.
// It first requests a shuffled list of connectors from the
// shuffler and then synchronizes many of them.
system.registerConnector(shuffler bind shuffled flatMap {
  case cs => manyOf(cs ++ sleepings : _*)
})

// Starts the system.
system.run()
```

```
  }
}
```

6.5.2 Expressivity

The above program, even though a bit more verbose than its equivalent Haskell version, is still quite concise. However, some of the constructs used are not really natural. Having to define an `act` function to perform a loop is a bit awkward. Since the `await` function never returns normally, it is pointless to call it within a loop, which explains why we had to resort to this trick.

The same exact problem arose within the context of actors in Scala[32]. To solve this syntactic problem, they introduced functions to mimic the native loops. If deemed to much of a problem, a similar approach could be taken here.

6.5.3 Final words

We conclude this chapter with this last example. We have seen how example applications could be written using either the Haskell framework or Scala framework. We discussed some points regarding the expressivity of the solutions and measured the running time of some examples for various parameters. Performing extensive benchmarks between the two frameworks and other concurrency paradigms is left as as future work.

Chapter 7

Discussion and Conclusion

7.1 Discussion

To conclude this thesis, we wish to discuss some points and make general remarks related to some aspects of this work.

7.1.1 Critics of the connectors

We begin our discussion by exploring some of the problems with connectors in their current version.

The importance of order of interactions

One difference between the formalisation of connectors and their implementation within the framework is that in the formalisation the interactions returned from the semantics are *unordered*. In the frameworks, the semantics function returns an *ordered stream*. This distinction is very important since interactions at the head of the streams will generally be computed for very little cost. On the contrary, interactions at the end of the stream will sometimes be prohibitively expensive to compute.

This also makes the commutativity of some connectors that hold in theory not true in practice. In the frameworks, the interactions at the start of the stream are in some manner *preferred* to those at the end. This point, which is not directly apparent in the frameworks and completely absent in the formalisation, could be further investigated.

Fairness

The previous remark on the fact that interactions are ordered also leads to fairness issues if no special care is taken. Indeed, the interactions at the head of the stream will generally be picked instead of to those later in the stream, and rightfully so since they are less expensive to compute. Since the order of interactions is generally fixed, the exact same interactions will tend to get executed, which may leave some atoms starving.

One way to alleviate this problem could be to randomly choose which branch of the `OneOf` combinator the semantics function will explore first. Each branch could be given different weights depending on the number of possible underlying interactions.

`Joined` and the distinction between computation and coordination

A different point, in some sense more conceptual, is the blurring of the distinction between coordination and computation brought by the `Joined` combinator. Indeed, this combinator is in some sense *too powerful*. This combinators enables atoms to specify to some extent the coordination of the system. Conceptually however, atoms should not have influence over the coordination, since they are part of the Behaviour layer, not the Glue layer.

7.1.2 Critics of atoms

Testing

The first remark we make about atoms is related to testing. This point, which has not been emphasised before, feels very important to make now. The approach of defining actors in isolation naturally leads to atoms that can be more easily tested. Indeed, for testing purposes, atoms could easily be run independently of the system in which they are used. Instead of receiving values from the other atoms, the testing framework could be used to send values on the port of the atoms in some specified order.

State machine representation

The second, and last, remark we make on atoms concerns the difference between the conceptual modelling of atoms in BIP and their implementation in the frameworks. Indeed, in BIP, the atoms are finite state machines, which have a definite behaviour. In the frameworks, atoms are free to perform anything and wait on any port at any time. Even though having more precise information on the potential future actions of an atom may prove useful for analysis purposes, we have decided to stick with the less restrictive approach in the framework for usability reasons.

One way we briefly investigated to alleviate this problem was the possibility to give users a way to optionally specify the underlying state machine of atoms. This information could be then verified at runtime and relied on for optimisation and analysis purposes.

7.1.3 General observations

Mixing with other paradigms

One point that we did not consider during this thesis is the mixing of the BIP model with other concurrency paradigms. We have reasons to believe that users of the framework will use it in combination with other models, as is common place with actors in

Scala[33] for instance. Indeed, it is sometimes easier to use other paradigms for some specific tasks, and users see little point in sticking to a single concurrency paradigm.

It is our personal opinion that this should not be rejected, but embraced. To do so, it should be ensured that the frameworks mix well with other libraries and do not make any assumption regarding the environment they are running on. Some libraries, bridging the gap between BIP and other concurrency models, such as actors and futures[34], could be developed.

7.1.4 Specificities of Haskell and Scala

In this last part of the discussion, we wish to make some observations related to the specificities of the different languages. The following are simply observations we have made during development and that we feel are relevant at this point.

Subtyping for easier combinations

The first point we look at is related to subtyping. One of the major difference between Haskell and Scala is that the latter supports subtyping. Due to subtyping, connectors are generally more easier to combine, especially when they are used in multiple contexts. This makes some trivial transformations that must be performed using `Mapped` and `ContraMapped` combinators in Haskell unnecessary.

Type inference

The above advantage that subtyping offers in terms of compatibility of connectors is counterbalanced by the limits of type inference in Scala. Since Scala only has local type inference, values will generally be inferred to be of very restrictive types when they are created, only to be unusable in later contexts. This problem is very apparent in the framework when ports are created. This forces the users of the framework to explicitly type the ports. This can be argued to be desirable, since it explicitly documents how ports can be used.

Purity

One of the very distinctive feature of Haskell is the enforced purity of functions. Has we have seen in the chapter on optimisations, having pure functions makes it possible to cache interactions. It also makes clear to the users of the framework which part of the framework can perform side effects.

On the other hand, Scala functions can perform any side effects when they are evaluated. The usual solution taken in Scala is to specify to users when pure functions are expected, and to leave them the responsibility to respect that contract made with the framework. Taking this approach, some optimisations, such as caching, could also be performed.

Continuations

An other point we wish to make concerning Haskell and Scala is about continuations. In the frameworks, as we have seen, the continuation of an atom must be obtained when it executes an `await` instruction. In Haskell, due to the way the `do`-notation is translated, this continuation is immediately obtainable without any burden on the users.

In Scala however, the continuation must be explicitly passed as an extra argument to the `await` function which cause some syntactic overhead. In addition, this means that `await` does not mix well with other constructs from the language, such as loops. A library of helper functions could be developed to lessen the problem.

Obscure operators

So far, we have discussed some issues related to the programming languages themselves. To end this section on Haskell and Scala, we wish to discuss a last point, not directly related to the languages, but to their standard library.

This specific problem is generally found in both Scala and Haskell and concerns the naming convention of operators. It is generally common place in those languages to name binary operators with sequences of symbols, which can sometimes be very obscure. In the framework, we avoided those obscure operators when possible. However, in Haskell, some of those functions are part of type classes, and their name can be very intimidating to users unfamiliar with Haskell. To alleviate this problem, operators such as `<*`, `*>`, `<>` and the like could be aliased and given domain specific names that novices in the language could use.

7.2 Future work

In this section, we discuss some of the future work that could be done to continue this work. In addition to some of the points we have already mentioned in the previous discussion, we would like to propose three different areas of research would could lead to some increase in the usefulness of the framework.

7.2.1 Caching

One idea that could be implemented as future work is the caching of interactions. This optimisation technique, which has already been discussed in some details in the chapter on optimisations, could potentially improve drastically the performance of BIP systems. This technique could also potentially be widely applied. Indeed, we believe that this technique could also be suited to other implementations of BIP.

In addition to possibly enhance the performance of systems, this optimisation also raises interesting research questions.

7.2.2 Analysis

One other aspect that could be investigated is the type of analysis that can be performed on systems described within the framework. Since the connectors are implemented as a tree-like structure, it could be possible to derive some interesting properties.

However, some of the connectors, such as `Joined`, due to their highly dynamic nature, may render any kind of analysis really difficult to perform. In addition to that, we have to recall that the behaviour of atoms is completely unknown. Combined together, those two aspects make systems really opaque to the framework.

We personally believe that, for any kind of analysis to be practically feasible, some more information should be obtainable from the description of system. We would recommend a deeper embedding of the different constructs within the host languages. Implementation techniques, such as *lightweight modular staging*[20], could be used in this context.

7.2.3 Distribution

A final area we want to mention before we finally conclude this thesis is the distribution of BIP systems. Indeed, distributed systems are becoming increasingly important, and abstractions to design distributed systems are needed. Previously, a lot of work has been performed on distributing BIP[27][28][29][30].

Furthermore, a lot of work also has been performed in functional programming languages research to allow the distribution of programs, with an emphasis on being able to communicate function closures over the network[35][36].

We believe that those new tools of functional programming languages could be used to implement a distributed version of the frameworks developed in this thesis.

7.3 Conclusion

In this thesis, we have presented the theory and implementation details behind the adaptation of BIP into two functional programming languages.

We have developed a theory of connectors based on combinators. This formalisation is expressive enough to take into account synchronisation, data transfers, priorities and dynamicity. Moreover, the connector combinators introduced in this thesis are shown to have many interesting algebraic properties. We also showed that obtaining interactions from a connector is a theoretically hard problem.

We then moved from theory to practice and discussed the implementation details of the two frameworks. We showed how the various theoretical concepts can be elegantly implemented in both Haskell and Scala. Even though the two implementations used different constructs that are idiomatic for each particular language, the general principles behind the implementation, as well as the architecture of the engine, are shared. We strongly believe that this approach could also be taken to implement BIP in other programming languages.

Then, to counter the intrinsic complexity behind the evaluation model of BIP, we discussed possible optimisations of the semantic function and of connectors. We also discussed the possibility of caching. We finally talked about early evaluation of stable interactions.

We then implemented example applications in both frameworks. This exposition showed the expressivity of the two frameworks. We were also able to measure performance of some particular examples.

With this thesis, we hope to have shown the following points:

- Connectors can be formalised as a algebra of combinators. This construction of connectors, which also models data transfers and priorities, follows nice algebraic properties and is very amenable to manipulation. This algebra also has the advantage of being directly transposable as a generalised algebraic data type.

- It is possible to embed BIP in functional programming languages. This embedding improves massively the usability of the language by allowing programmers to use higher level constructs to describe their systems. The generic abstractions offered by the language can also be applied to describe connectors in a concise way.

- It is possible to describe complex concurrent systems in Haskell and Scala by separating the computation code and the coordination code. This separation leads to systems that are easier to understand and to test, while still being reasonably performant.

Bibliography

[1] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[2] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software,"

[3] E. W. Dijkstra, *Cooperating sequential processes.* Springer, 2002.

[4] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[5] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[6] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pp. 3–12, Ieee, 2006.

[7] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE software*, vol. 28, no. EPFL-ARTICLE-170496, pp. 41–48, 2011.

[8] S. L. P. Jones, *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[9] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "The scala language specification," 2004.

[10] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "D-finder: A tool for compositional deadlock detection and verification," in *Computer Aided Verification*, pp. 614–619, Springer, 2009.

[11] S. Bliudze, A. Mavridou, R. Szymanek, and A. Zolotukhina, "Coordination of software components with BIP: application to OSGi," in *Proceedings of the 6th International Workshop on Modeling in Software Engineering - MiSE 2014*, (New York, New York, USA), pp. 25–30, ACM Press, 2014.

[12] S. Bliudze and J. Sifakis, "The Algebra of Connectors — Structuring Interaction in BIP," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pp. 11–20, ACM, 2007.

[13] A. Basu, S. Bensalem, M. Bozga, P. Bourgos, and J. Sifakis, "Rigorous system design: the bip approach," in *Mathematical and Engineering Methods in Computer Science*, pp. 1–19, Springer, 2012.

[14] G. A. Agha, "Actors: a model of concurrent computation in distributed systems," 1985.

[15] S. Bliudze, M. Bozga, M. Jaber, and J. Sifakis, "Architecture Internalisation in BIP," in *Proceedings of the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering*, pp. 169–178, ACM, 2014.

[16] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, pp. 358–366, 1953.

[17] S. Awodey, *Category Theory*. Oxford Logic Guides, OUP Oxford, 2010.

[18] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 60–76, ACM, 1989.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[20] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Acm Sigplan Notices*, vol. 46, pp. 127–136, ACM, 2010.

[21] J. Launchbury and S. L. Peyton Jones, "Lazy functional state threads," in *ACM SIGPLAN Notices*, vol. 29, pp. 24–35, ACM, 1994.

[22] H. Apfelmus, "The operational monad tutorial," *The Monad. Reader*, vol. 15, pp. 37–55, 2010.

[23] E. A. Kmett, "profunctors: Profunctors." `http://hackage.haskell.org/package/profunctors`, July 2011.

[24] C. McBride and R. Paterson, "Applicative programming with effects," *Journal of functional programming*, vol. 18, no. 01, pp. 1–13, 2008.

[25] P. Wadler, "Monads for functional programming," in *Advanced Functional Programming*, pp. 24–52, Springer, 1995.

[26] S. P. Jones, A. Gordon, and S. Finne, "Concurrent haskell," in *POPL*, vol. 96, pp. 295–308, Citeseer, 1996.

[27] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "From high-level component-based models to distributed implementations," in *Proceedings of the tenth ACM international conference on Embedded software*, pp. 209–218, ACM, 2010.

[28] S. Bensalem, M. Bozga, J. Quilbeuf, and J. Sifakis, "Knowledge-based distributed conflict resolution for multiparty interactions and priorities," in *Formal Techniques for Distributed Systems*, pp. 118–134, Springer, 2012.

[29] J. Quilbeuf, *Distributed Implementations of Component-based Systems with Prioritized Multiparty Interactions. Application to the BIP Framework.* Theses, Université de Grenoble, Sept. 2013.

[30] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "A framework for automated distributed implementation of component-based models," *Distributed Computing*, vol. 25, no. 5, pp. 383–409, 2012.

[31] R. Milner, *Communication and concurrency.* Prentice-Hall, Inc., 1989.

[32] P. Haller and M. Odersky, "Actors that unify threads and events," in *Coordination Models and Languages*, pp. 171–190, Springer, 2007.

[33] S. Tasharofi, P. Dinges, and R. E. Johnson, "Why do scala developers mix the actor model with other concurrency models?," in *ECOOP 2013–Object-Oriented Programming*, pp. 302–326, Springer, 2013.

[34] H. C. Baker Jr and C. Hewitt, "The incremental garbage collection of processes," *ACM SIGART Bulletin*, vol. 12, no. 64, pp. 55–59, 1977.

[35] J. Epstein, A. P. Black, and S. Peyton-Jones, "Towards haskell in the cloud," in *ACM SIGPLAN Notices*, vol. 46, pp. 118–129, ACM, 2011.

[36] H. Miller, P. Haller, and M. Odersky, "Spores: A type-based foundation for closures in the age of concurrency and distribution," tech. rep., 2014.

[37] B. O'Sullivan, "criterion: Robust, reliable performance measurement and analysis." `https://hackage.haskell.org/package/criterion`, October 2014.

[38] S. P. Jones, "Beautiful concurrency," 2007.

[39] B. Yorgey, "Typeclassopedia," *The Monad.Reader*, pp. 17–68, Mar. 2009.

[40] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 12–1, ACM, 2007.

[41] A. Kennedy and C. V. Russo, "Generalized algebraic data types and object-oriented programming," in *ACM SIGPLAN Notices*, vol. 40, pp. 21–40, ACM, 2005.

[42] H. Søndergaard and P. Sestoft, "Referential transparency, definiteness and unfoldability," *Acta Informatica*, vol. 27, no. 6, pp. 505–517, 1990.

Appendices

Appendix A

Introduction to Category Theory

In this appendix, we present the definition of several notions from Category Theory[17] that have been applied to the context of connector combinators.

A.1 Category

Definition 5 (Category)**.** A *category*[17] is defined as a collection of *objects* and *arrows*. Each arrow has a *source* object and a *target* object. We denote by $f : a \to b$ the fact that the arrow f has for source an object a and for target an object b. The objects and arrows must obey the following laws:

Composition There exists for any two arrows $f : a \to b$ and $g : b \to c$ an arrow for their composition, denoted by $g \circ f : a \to c$. The composition is associative, that is, for any arrows $f : a \to b$, $g : b \to c$ and $h : c \to d$, we have that:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

Identity There exists for every object x an identity arrow $1_x : x \to x$. Every identity arrow is the identity with regards to arrow composition. That is, for any arrow $f : a \to b$:

$$f \circ 1_a = 1_b \circ f = f$$

A.2 Functor

Definition 6 (Functor)**.** A *covariant functor*[17], or simply *functor*, F is mapping between two categories C and D such that the following hold:

Mapping of objects For any object x of the category C, F associates an object $F(x)$ of the category D.

Mapping of arrows For any arrow $f : a \to b$ of C, F associates an arrow $F(f) : F(a) \to F(b)$ in D, such that the following properties hold:

1. Identities are preserved by F, that is $F(1_x) = 1_{F(x)}$.
2. Composition is preserved by F, that is $F(g \circ f) = F(g) \circ F(f)$.

Remark. The identity function that maps arrows and objects of a category to themselves is a functor. We denote the identity functor on a category C by 1_C.

Remark. Given a functor F between category C and D and a functor G between categories D and E, their composition, denoted by $G \circ F$, is a functor between categories C and E.

Definition 7 (Contravariant functor)**.** A *contravariant functor*[17] G is a mapping between two categories C and D such that the following hold:

Mapping of objects For any object x of the category C, G associates an object $G(x)$ of the category D. This is exactly the same as for covariant functors.

Mapping of arrows For any arrow $f : a \to b$ of C, G associates an arrow $G(f) : G(b) \to G(a)$ in D, such that the following properties hold:

1. Identities are preserved by G, that is $G(1_x) = 1_{G(x)}$.
2. Composition is reversed by G, that is $G(g \circ f) = G(f) \circ G(g)$.

Note that the composition arrow is reversed compared to the covariant functor

A.3 Natural transformation

Definition 8 (Natural transformation)**.** Given two covariant functors F and G between categories C and D, a *natural transformation*[17] $\eta : F \to G$ is a family of arrows that satisfies the following properties:

1. For each object x of C, an arrow η_x, called the *component* of η at x, exists between the objects $F(x)$ and $G(x)$ of the category D.

$$\eta_x : F(x) \to G(x)$$

2. For any arrow $f : x \to y$ in the category C, the following holds:

$$\eta_y \circ F(f) = G(f) \circ \eta_x$$

A.4 Monad

Definition 9 (Monad)**.** A functor F from a category C to itself and a pair of natural transformations η and μ such that:

$$\eta : 1_C \to F$$
$$\mu : F \circ F \to F$$

form a *monad* if the following hold for all objects x of the category C:

$$\mu_x \circ F(\mu_x) = \mu_x \circ \mu_x$$
$$\mu_x \circ F(\eta_x) = \mu_x \circ \eta_x = 1_x$$

The natural transformation η is called *unit* and the natural transformation μ is called *join* or sometimes *multiply*.

Appendix B

Introduction to Haskell

The first language in which we implement BIP as a DSL is Haskell, a functional programming language. In this appendix, we will introduce the key concepts and syntax of Haskell. This chapter doesn't have the pretension to present a complete view of Haskell. However, the key points shown here should be sufficient to get a good understanding of the specificities of the language and to tackle the details of our implementation.

B.1 Basic syntax

Haskell, as a functional programming language, has a strong focus on expressions and functions. The following shows the basic syntax for expressions, most of which involving functions.

Function application

Below is shown the way a function f is called with an argument x.

```
f x
```

When multiple arguments are fed to a function, they are separated by a space. For instance, below is shown the way to call a function f with three arguments x, y and z.

```
f x y z
```

Note that arguments of a function may be functions themselves.

Function composition

Function composition is denoted by a single `.` in Haskell. For instance, the function that would first apply f and then g could be written as:

```
g . f
```

Operators

In Haskell, operators, such as `+`, `-` and `*` are normal functions, except that they must be used in infix position. For instance, the sum of two values x and y can be written as:

```
x + y
```

Operators can also be used in prefix position just as other function if they are surrounded by a pair of parenthesis. The following thus also denotes the sum of x and y:

```
(+) x y
```

Note that function composition itself is an operator. The composition of two functions f and g can therefore also be written as:

```
(.) g f
```

Precedence and parenthesis

Operators are given their natural precedence level and associativity. Naturally, parenthesis can be used to change the natural associations. For instance, the following to expressions are equivalent:

```
1 + 2 * 3
1 + (2 * 3)
```

Which are different from the expression:

```
(1 + 2) * 3
```

It is current practice in Haskell to avoid using too many parenthesis using the function application operator `$`. The operator is defined as:

```
f $ x = f x
```

This apparently useless operator, by having a conveniently low precedence level, allows programmers to avoid writing parenthesis at the end of an expression. For instance, the following two expressions are equivalent:

```
h $ g $ f x y
h (g (f x y))
```

B.2 Evaluation strategy

Haskell is *lazy* by default language. Which means that, contrary to most programming languages which are *strict* and evaluate the arguments of a function before the function is actually called, the arguments of a function are only evaluated if and when they are needed.

This evaluation strategy, called lazy evaluation, or call-by-need, was a design decision taken from the start[40]. This decision forced the language to control side effects and enfore *purity*.

B.3 Purity

Due to lazy evaluation, all Haskell functions are *pure*, meaning that they can not have any side effect. The application of the same arguments to a function will thus always lead to the same result. This property is called *referential transparency*[42].

This also means that in Haskell all values are immutable. There is no concept of mutable variable in the language itself. We will see later in this section how side effects are introduced back in the language, in a controlled manner.

B.4 Declarations

At the top level, Haskell programs consist of possibly many constants and functions *declarations*. Those declarations bind a name to an expression. Those declarations can be (mutually) recursive.

Below are two example declarations:

```
pi = 3.14

fact x = if x <= 0 then 1 else x * fact (x - 1)
```

Note that the declarations are immutable, meaning that there are no ways to change a declaration during lifetime of a program. All declarations are thus constant.

Pattern matching

Generally, functions will need to inspect their arguments and return a value accordingly. A common way to inspect the arguments received by a function is called *pattern matching*. A function declared using pattern matching specifies a list of *patterns* which specify the shape the arguments must have. For instance, here is how a function that computes the length of a list could be defined:

```
length [] = 0
length (x : xs) = 1 + length xs
```

In the above declaration, two patterns are defined. The first pattern only matches the empty list [] and the second any non-empty list. Notice that patterns, such as the second pattern in the example, may contain variables. Those variables will be bound to the actual values when the function is called. For instance, in the second pattern of the length function, x will be bound to the value at the head of the list and xs will be bound to the rest of the list. When, as in the previous example, some variables are not needed in the right hand side of the declaration, they can be replaced by an underscore placeholder in the pattern.

```
length [] = 0
length (_ : xs) = 1 + length xs
```

Pattern matching using case

Pattern matching can also be performed within the body of an expression, by using the case construct, as demonstrated by the following example:

```
map f xs = case xs of
    [] -> []
    y : ys -> f y : map f ys
```

B.5 Typing

Haskell is a strongly typed language. All expressions in Haskell have a type, which can be inferred by the compiler or explicitly stated in the program using a type signature. Normally, all top level declarations have an explicit accompanying type signature, even though they are most of the time not strictly needed. For instance, here are some of the previous example declarations, this time with explicit type signatures:

```
pi :: Double
pi = 3.14

fact :: Integer -> Integer
fact x = if x <= 0 then 1 else x * fact (x - 1)

length :: [a] -> Integer
length [] = 0
length (x : xs) = 1 + length xs
```

Standard types

The following types are part of Haskell and its standard library.

Name	Type	Values
Integers	`Int`	Machine integers
Integers	`Integer`	Unbounded integers
Floating point numbers	`Float`	Single precision
Floating point numbers	`Double`	Double precision
Booleans	`Bool`	`True` and `False`
Functions	`a -> b`	Functions from `a` to `b`
Lists	`[a]`	Lists of `a`
Tuples	`(a, b)`	Pairs of `a` and `b`
Optional values	`Maybe a`	`Just a` or `Nothing`
Unit	`()`	The single value `()`

Contrary to many programming languages, Haskell has no notion of sub-typing.

Polymorphism

Notice that some functions have generic types, such as for instance the length function we previously defined.

```
length :: [a] -> Integer
```

Those functions (or even values) are called polymorphic. They inhabit more than one type. Polymorphic functions are common place in the Haskell standard library and in Haskell in general.

B.6 Algebraic data types

So far we have only seen types that were already defined in the standard library. In addition to those already existing types, Haskell programmers can define their own types and data structures. In Haskell, *Algebraic Data Types* are used by programmers to define their own datatypes. An algebraic data type consists of possibly many *constructors*, each of which can have many fields.

A good example of an algebraic data type is the binary tree. The following data declaration defines a binary tree containing integers as a algebraic data type:

```
data Tree = Node Int Tree Tree
          | Empty
```

This declaration states that trees are either nodes containing an integer value and two subtrees, or empty trees. Note that algebraic data types can be recursively defined, as the example shows.

The three following expressions are examples of values of the type `Tree` we have just defined:

```
Empty
Node 1 Empty Empty
Node 1 (Node 2 (Node 3 Empty Empty) Empty) (Node 2 Empty Empty)
```

Parameterised algebraic data types

Algebraic data types can also be parameterised by one or many other types. In the above `Tree` data type, the type of values was restricted to `Int`. The following definition of `Tree` makes it possible to create trees holding values of any given type `a`.

```
data Tree a = Node a Tree Tree
            | Empty
```

The following two expressions are then of type `Tree Int`:

```
Empty
Node 1 Empty Empty
```

And the following two of type `Tree Bool`:

```
Empty
Node True Empty (Node False Empty Empty)
```

Generalised algebraic data types

Generalised algebraic data types[41] (GADTs), are an extension of algebraic data types that allow constructors to specify on which type parameters they are defined. The basic example of a GADT is the definition of a minimalistic expression language.

```
data Expr a where
    Literal  :: a -> Expr a
    Addition :: Expr Int  -> Expr Int -> Expr Int
    Equal    :: Expr Int  -> Expr Int -> Expr Bool
    If       :: Expr Bool -> Expr a   -> Expr a    -> Expr a
```

The above definition statically ensures that, for instance, addition is only performed between two expressions that evaluate to integers, and also that addition evaluates to an integer itself. Using non-generalised algebraic data type, we would have no way to restrict the type of expressions constructed by the `Addition` constructor for instance.

GADTs will prove useful for implementing some of the data types used in the Haskell version of the BIP framework.

B.7 Type classes

An other very important and singular aspect of Haskell is the concept of *type classes*[18]. A type class, parameterised by a type a, defines constants and functions that must be defined by all types a that are *instances* of the type class. Those constants and functions thus become polymorphic: they are members of all types that are instances of the type class.

The following type class for instance defines a unique function, ==, which indicates that two values are equal.

```
class Eq a where
    (==) :: a -> a -> Bool
```

Every type on which equality is defined can then instantiate the type class. One can find for example in the standard library Eq instances for many types. The following exemplifies how to instantiate a type class.

```
instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False
```

This instance specifies that True and False are equal to themselves and to nothing else.

Constraints

Type classes can appear in the type signature of an expression to constraint some of the type parameters. For instance, if we were to define a polymorphic function that checks that two values of some type a are different, we might want to ensure that the type a supports equality. This is performed as follows:

```
different :: Eq a => a -> a -> Bool
different x y = if x == y then False else True
```

The above type signature specifies that different is binary function defined on all type a which support equality.

The contraints can also appear in instances declaration to restrict the types of which the instance is defined. For instance, here is how equality on Maybe may be defined:

```
instance Eq a => Eq (Maybe a) where
    Nothing == Nothing = True
    Just x  == Just y  = x == y
    _       == _       = False
```

Similarly, constraints can also appear in the class declaration. For instance, the class `Ord`, which states that elements of a type can be ordered, requires that equality is defined on the type.

```
class Eq a => Ord a where
    (<=) :: a -> a -> Bool
```

Standard type classes

The Haskell standard library includes may different type classes and instances. The following are some basic examples.

Eq the class of types whose values support equality.

Ord the class of types whose values can be linearly ordered.

Show the class of types whose values can be converted to strings.

The standard library contains also many other, higher order type classes, which will be of special interest to us. The concepts can be difficult to grasp at first, and the very small introduction we give to them here might not be sufficient. A more complete and intuitive introduction can be found on the *typeclassopedia*[39][1].

Functor the class of type constructors which can be mapped over. The type class defines a unique function: `fmap`.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

This class can be thought of as the class of "containers" that can be mapped over. For instance, lists are an instance of this type class. It is indeed possible to apply a function to all elements of a list and collect all of the results in a new list.

The name of the type class comes from the concept of functors in category theory, which is closely related.

Monad is a super class of `Functor`. In addition to support mapping over the elements with `fmap`, instances also support the transformation of a normal value into a value contained in the type (using `return`), and *sequencing* (using `>>=`), as we will see.

```
class Functor m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

[1] The *typeclassopedia* is currently available on the website `https://www.haskell.org/haskellwiki/Typeclassopedia`.

Lists are a good example of a monad. Indeed, `return` corresponds to wrapping the value in a singleton list. The function `>>=` corresponds to `concatMap`, also know as `flatMap` in some other languages. This function applies a function returning a list over all elements and then concatenates the results.

```
instance Monad [] where
    return x = [x]
    [] >>= _ = []
    (x : xs) >>= f = f x ++ (xs >>= f)
```

This type class, as the previous, is named after a concept from category theory, the monad.

Monads are of particular interest since they are a good abstraction for computations with effects, as we will see in the next section.

B.8 Computations with effects

Haskell, being a pure and lazy language, enforces a strong separation between evaluation of expressions and execution of computations with side effects. So far, we have only seen examples of pure functions, we can not have any side effects. However, to be useful, a program must have side effects. Otherwise, it would be impossible to even print the result of a computation to the console!

B.8.1 The `IO` type constructor

The solution adopted by Haskell is to have a type constructor, called `IO`, which represents computations that may have side effects. A value of type `IO Int` therefore represents a computation which results in an integer value. Here are some examples of such instructions present in the standard library:

```
-- Prints a string to the console.
putStrLn :: String -> IO ()

-- Gets a line from standard input.
getLine :: IO String
```

It is important to note that the creation of a value of type `IO a` does not mean that the action described by the value will be performed! For instance, the evaluation of `putStrLn "Hello World"` will *not* produce any output. Passing a value to `putStrLn`, when evaluated, will simply create an action that, when executed, would print something.

B.8.2 The main action

Only a *single* action of type `IO ()` gets executed, the one at the entry point of the program, named `main`.

```
main :: IO ()
main = putStrLn "Hello World"
```

The execution of the above program will indeed show a message to the console, as would be expected. As we have just said, it is only possible to execute a single action per program, the `main` action. To have more interesting programs, we must thus have a way to *combine* actions into some more complex ones. This is were the `Monad` type class comes into play.

B.8.3 The Monad instance of IO

The monad instance of `IO` gives a way to sequence several `IO` actions together. For instance, here is how to combine the `getLine` instruction with the `putStrLn` function to obtain a function that reads a line from standard input and print it to standard input:

```
main :: IO ()
main = getLine >>= putStrLn
```

The do-notation

The `do`-notation, a syntactic sugar of Haskell, allows programmers to chains actions more easily without explicitly using the `>>=` function. The previous example could be simply rewritten as:

```
main :: IO ()
main = do
    x <- getLine
    putStrLn x
```

B.9 Concurrency support

The Haskell language and runtime has support for concurrency, mainly through two concepts: `forkIO` and `MVars`[26]. Other concurrency libraries, such as `STM`[38], will not be exposed here.

B.9.1 forkIO

The instruction `forkIO` allows programmers to fork a new thread to execute some action.

```
forkIO :: IO () -> IO ThreadId
```

The threads spawned by this instruction are not OS threads, but so-called *lightweight* threads, which are handled by the Haskell runtime. Those threads have very low overheads associated to context switches and creation.

B.9.2 MVars

MVars are mutable variables that have been designed to work in concurrent settings. An MVar a can either be either be empty, or contain a value of type a. MVars support, amongst others, the following operations:

```
-- Creates a new variable
newMVar :: a -> IO (MVar a)

-- Takes the value from a variable.
takeMVar :: MVar a -> IO a

-- Puts a value in a variable.
putMVar :: MVar a -> a -> IO ()
```

All the above operations are *atomic*, meaning that they appear to take place at a single indivisible point in time. In addition, trying to take the value of an empty variable and trying to put a value in an already full variable will block until the operation is possible. The implementation makes sure that threads are not blocked indefinitely on the variable, given that the variable is not hold by a thread indefinitely.

Contact information

Address

Romain Edelmann
Rue des Près-du-Lac 7b
1400 Yverdon-les-Bains
Switzerland

E-mail address

romain.edelmann@gmail.com