Session types in Cloud Haskell
Thesis

Ferdinand van Walree
ICA-3874389

Supervisors:
Wouter Swierstra
Andres Löh

Department of Computing Science
University of Utrecht

September 4, 2017

Contents

Abstract

The implemention of a communication protocol is susceptible to human errors. Two processes engaging in such a protocol can deadlock or crash if their implementation of the protocol is faulty. In the context of Cloud Haskell, a distributed programming library for Haskell, we address this problem by providing a library for annotating Cloud Haskell programs with session types. A session type describes a protocol and by annotating a program with a session type we produce a static guarantee that the program correctly implements the protocol. We demonstrate how we use a deep-embedded domain-specific language for writing session typed programs to define several interpreters that can evaluate session typed programs. The most important interpreter is the one that evaluates a session typed program to Cloud Haskell semantics. Other interpreters can purely evaluate a session typed program, interactively step through a session typed program, visualize a session type using a diagram and normalize session typed programs to normal form. We introduce an optimization that reduces the amount of administrative communication required for n-ary nested branching session types. Further, we also introduce an optimization that turns the quadratic complexity of any session typed program, caused by left-associatively nested binds, into linear complexity using the codensity monad. To conclude, we measured the performance overhead that our library introduces compared to that of the Cloud Haskell library.

1 Introduction

Protocols exist to ensure that two processes have a structured way of communicating. The protocol specifies what type of message should be sent by which process and at what time. A protocol is usually specified in an ordered list of sends to be completed by either one of the processes. An example of a simple ping pong protocol can be seen below.

```
A: send "Ping"
B: send "Pong"
```

Process `A` sends a message containing the string `"Ping"`. It is assumed that this message is then received by process `B`. The next action is then for process `B` to send back the string `"Pong"`. The `"Pong"` message is essentially a confirmation by `B` that it received the `"Ping"` message from `A`.

Implementing such a protocol in a programming language does not only consist of sending a message, but also receives have to be specified. Since the protocol itself doesn't specify when a receive should occur the programmer is allowed to do a receive at any point during the program. Though, it should be quite obvious to the programmer that, in the case of process `A`, the receive should only be done after the send. Still it is possible for the programmer to make a mistake, especially if a protocol is more complex it becomes easier to make such mistakes. For example, the code snippet below is an incorrect implementation of the above protocol (we will now include the receives).

```
A: recv
A: send "Ping"
B: recv
B: send "Pong"
```

Process `A` starts with a receive, but this immediately causes the process to become stuck as process `B` is also waiting for a message. Consequently, the protocol cannot be continued.

Another mistake that can be made is for process `A` or `B` to send the wrong message.

```
A: send "Pong"
B: recv
B: send "Pong"
A: recv
```

Again, the implementation of process `A` is incorrect as it sends the string `"Pong"` instead of `"Ping"`. It could very well be that `B` does not inspect the message it received and

sends back a verification no matter what, allowing the protocol to continue. However, that still does not make the implementation of the protocol correct. Process `A` should send `"Ping"` and process `B` should receive a string `"Ping"`.

Both mistakes are not in the protocol itself, but in the implementation of the protocol. Since the protocol is statically known we could use this information to also statically enforce a correct implementation. A solution within the domain of typed languages is to use session types [7]. Session types encode protocols as types, where this type describes a sequence of actions that must be completed in that exact order. For example, we could write a session type that describes the ping pong protocol.

```
Ping :!> Pong :?> Eps
```

The session type says that first `Ping` must be sent, followed by receiving a `Pong` message after which the session type ends. The session type does not describe the entire protocol. It only describes a protocol in the perspective of one participating process. So the above session type would correspond to what process `A` must do to complete the protocol and the following session type describes the actions of process `B`.

```
Ping :?> Pong :!> Eps
```

This session type is the dual of the session type corresponding to process `A`. The dual of a session type replaces an action with its opposite; a send is replaced with a receive and a receive is replaced with a send. Using session types to describe protocols allows us to statically enforce sending and receiving of messages in the correct order and of the right type. If a program's implementation does not conform to its session type, a type error should be raised. This rules out any programs that give an incorrect implementation of the specified protocol. Then the two previously mentioned mistakes are no longer possible to make.

A domain that could benefit from using session types is distributed programming. In distributed programming there are two or more processes that do work on one or more computers. On which exact computer each process is located should not matter. Usually a library for distributed programming exposes primitives that can be used to send messages to a process given some data type that identifies the process. One such library is Cloud Haskell [6] that can be used to write distributed programs in Haskell. Cloud Haskell runs processes on a node. A node is an abstraction from the hardware and generally a single node is used for each computer, but if necessary more can be used. The Cloud Haskell library allows processes to communicate with each other, but also suffers from the problems described above. If two processes don't correctly implement a protocol they might become stuck and depending on how the distributed network is programmed this could have catastrophic consequences for the entire network. We can use session types to resolve these problems for Cloud Haskell.

Several session type libraries for Haskell [9, 13, 15, 16] have been made already. Usually these libraries work independently and combining it with an existing library can be

difficult. We will aim to write another session types library, that can be combined with existing libraries, such as Cloud Haskell, while using recent extensions to GHC to further improve the session type implementation compared to those in the existing session type libraries. As such this master's thesis will make the following contributions:

- We begin by giving a more in depth description of Cloud Haskell (section 2). This thesis will show how we can use session types in combination with Cloud Haskell. We therefore often make use of Cloud Haskell semantics. That makes it important for the reader of this thesis to at least have a basic understanding of how Cloud Haskell works.

- Next we give a definition for session types (section 3) and discuss how session types should be exposed at term level. We identify two approaches for using a DSL to do so. We decide on using a deep embedding for a DSL, such that we are not limited to only Cloud Haskell semantics. Thereby making our session types library more generic. Finally we define a deep embedded DSL, expose an API and implement the Cloud Haskell semantics.

- We further improve the combination of session types in Cloud Haskell (section 4), by exposing more functions from the Cloud Haskell library in the session type API. We also solve several problems that were stopping us from writing fully functional session typed programs that can be called from a Cloud Haskell problem. We also introduce a notion of a session typed channel that can be used to transmit messages of different types and that can also handle branching and recursion of types.

- We define several interpreters for our DSL (section 5) that have different semantics: generating a pure model of the behavior of a session typed program, interactively evaluating a session typed program, visualizing session types in a diagram and normalizing session typed programs to a normal form.

- We apply two optimizations to our session typed library. The first optimization tackles the inefficiency in the Cloud Haskell semantics for branching session types. The second reduces the quadratic complexity of the repeated use of bind to linear complexity by using the codensity monad.

- Finally, we perform benchmarking where we compare a program written using the Cloud Haskell library to a session typed program with the same sematics. We also compare how the different optimizations affect the performance.

2 Cloud Haskell

In the previous section we have already given a short introduction of Cloud Haskell as a distributed programming library for Haskell. In this chapter we will go a little bit more in depth, but most of all describe several primitives that are necessary to know in the following chapters.

Cloud Haskell is inspired by Erlang [19]. Each Erlang runtime system is a node and each Erlang program can spawn processes both local and remote. Instead of shared variables, Erlang makes use of message passing for communication between processes. A node receives a message and inserts it into a message queue of the process that the message should be forwarded to. It does not matter where the message came from, even if the process had sent the message to itself will it still be inserted in the message queue. The process may then access his message queue to retrieve the message.

Cloud Haskell defines a data type `Node` that corresponds to nodes in Erlang. A `NodeId` is associated to this `Node` that consists of an IP address and a port. Unlike Erlang, a single node does not correspond to a single Haskell runtime system. We can start multiple nodes on different Haskell threads. For processes there is the `Process` monad, which is run on a Haskell node. A process is essentially a lightweight thread that has its own piece of memory and acts as the interface of the library. Initially a single process is run on a node, but more can be spawned to run on this node from that process. A process can also spawn a process on a remote node if it has access to its `NodeId`. A process itself is identified locally by an integer, located in the `ProcessId` of the process, and remotely by the same data type plus the `NodeId` of the node that runs it.

Cloud Haskell exposes functions to start a new node and run a process on top of that node:

```
newLocalNode :: Transport -> IO LocalNode
runProcess :: LocalNode -> Process () -> IO ()
```

The `Transport` type comes from the *network-transport* library and serves as the backend that facilitates the actual transmission to and from nodes. After starting a node a `LocalNode` is returned that can be used to control a node. For example, the `runProcess` function runs a given `Process` computation on top of a node by passing the corresponding `LocalNode`. Processes are managed using the `Process` monad. Using it we can send or receive messages, start new processes, kill the current process and more.

In untyped messaging, messages are sent by a `Process` to a given `ProcessId`.

```
send :: Serializable a => ProcessId -> a -> Process ()
```

The function `send` takes two arguments. First is the `ProcessId` of a process to which the message should be sent. Second is the content of the message. Since this message should be polymorphic in the type of the content, a `Serializable` constraint is applied to this type. The `Serializable a` constraint allows us to serialize and deserialize values of type `a`.
Once a message has been sent, it is received (assuming it arrives) by a `Node`. It inspects the type of message and based on that chooses where and how to forward this message. In the case of untyped messaging, a message is forwarded to the message queue of the associated `Process`. Messages are stored in a data type `Message` that existentially quantifies the type of the actual value. This allows the message queue to store values of different types. The function `expect` allows one to retrieve a message from the message queue.

```
expect :: Serializable a => Process a
```

When retrieving a message, it is required for the programmer to specify the type of the message that should be retrieved from the message queue. The message queue is traversed until it finds a message of the right type at which point extracts it from the message queue and returns it. If none is available the function will block until a message arrives.

Cloud Haskell also provides the option to use typed channels for sending and receiving of messages. A typed channel gives a static guarantee that the receiving end of a channel knows how to deal with the messages it receives. There exist three primitive functions for dealing with typed channels.

```
newChan :: Serializable a => Process (SendPort a, ReceivePort a)
sendChan :: Serializable a => SendPort a -> a -> Process ()
recvChan :: Serializable a => ReceivePort a -> Process a
```

A channel consists of a tuple of a send port and a receive port. Both are parameterized by a type variable denoting the type of the messages that can be transmitted through the channel. That means a channel, unlike untyped messaging, can only send and receive messages of exactly one type. The send port contains information about where the channel was created (ip address, port and channel identifier). The function `sendChan` uses this information to send a message to a node of that location. Upon receiving the typed messaged the node stores the message in the corresponding `TQueue`. Finally, a process may read from a channel using `recvChan`. A `ReceivePort` merely contains the channel identifier, which is used to read from the corresponding `TQueue`. If the queue is empty the function blocks, otherwise the messages are extracted in a FIFO order. You cannot serialize a `ReceivePort`, because it corresponds to a specific `Process`. On the other hand a `SendPort` can be serialized. It only contains information of where a message should be sent to, which is why why more than one processes may use the send port.

One important feature of distributed programming is the ability to manage threads. It should be possible to run code on different processes or nodes, monitor the activity of these processes, kill processes and more.

```
-- Simplified
spawnLocal :: Process () -> Process ProcessId
spawn :: NodeId -> Process () -> Process ProcessId
monitor :: ProcessId -> Process MonitorRef
kill :: ProcessId -> String -> Process ()
```

The function `spawnLocal` starts a new process on the local node. Its second argument is the code that should be executed. It also returns the `ProcessId` of the newly created process allowing the two processes to communicate. There is also `spawn` that spawns the process on the node that corresponds to the given `NodeId`. The `monitor` function takes a `ProcessId` of the process that it should monitor. It returns a `MonitorRef`, which can be used to disable monitoring. A process that monitors another process may receive notifications that say that something happened to the monitored process. Usually that means the monitored process has died for some reason. The last function can be used to forcefully kill a process.

The Cloud Haskell library defines many more combinators for dealing with messaging and managing processes or nodes, but it is not the purpose of this paper to provide a full description of the Cloud Haskell API. In the next sections we will try to extend the above mentioned functions by decorating them with session types.

3 Session Types

The introduction sketched an example of how session types might be encoded in Haskell. This section will present how session types should be defined and how we can bind Cloud Haskell semantics to each session type. In the first section we construct a very simple session type, define a domain specific language (DSL) for session typed programs, write a simple API for it and show how we can evaluate a session typed program. Then in the sections after that we will improve the DSL and add two new session types.

3.1 Send and receive

3.1.1 Type representation

We will start off by defining a data type for session types that can encode a send, receive and the empty session type. We do this by defining a data type with three constructors.

```
infixr 5 :!>
infixr 5 :?>
data ST = (:!>) Type ST | (:?>) Type ST | Eps
```

The first constructor of `ST` represents the 'send' session type. It takes an argument of kind `Type` [1] and another argument of kind `ST`. The first argument represents the type of the value that we are sending and the second argument is the continuation of the session type, i.e. we might expect another send session type to follow or we could end the session type.
The second constructor of `ST`, `:?>`, represents the receiving session type. Like `:!>` it takes arguments of the same kind. The only difference is that the first argument represents the type of a value that we are receiving.
Finally, the third session type, `Eps`, is the empty session type. It has no arguments and using it in a session type means that we do not want any session typed operation to occur after it, meaning that a protocol is finished once we reach an `Eps`.

We will now construct a session type that represents the ping-pong protocol from the introduction.

```
data Ping = Ping
data Pong = Pong

Ping :!> Pong :?> Eps
```

[1] The kind `Type` is the same as kind $*$

We use two data types to represent the a ping message and a pong message. First we declare that the protocol must send a `Ping` using `:!>`. Then we proceed with the continuation of the send session type, its second argument, that says that we must receive a a message of type `Pong`. Finally the protocol closes with `Eps`.

Unlike existing work by Puecella & Tov [15], we have chosen to define session types using a single data type instead of having a data type for each session type construct. Because of extensions added to GHC we are now able to promote data types [23]. This means that the type constructor of a data type becomes a kind and the data constructors of a data type become types. Now it should also make sense why `:!>` and `:?>` take types of a specific kind instead of values of a specific type. The benefit of using a single data type and promoting it is that it becomes impossible to define invalid and nonsensical session types. For instance we could modify the ping-pong example slightly to be left associative.

```
(Ping :!> Pong) :?> Eps
```

This example is ill-typed for two reasons. First of all the second argument to `:!>` has kind `Type`, but it must have kind `ST`. The second reason is that the first argument to `:?>` is not of kind `Type`. By promoting `ST` we were able to disallow constructing invalid session types.

For every valid session type there is also a dual session type. The dual of a session type replaces its session type constructors with session type constructors that represents the opposite action. For example, the send session type `:!>` represents sending a value. The opposite of sending a value is receiving one. Hence the dual of `:!>` is `:?>`. The duality property ensures that two processes in a session can not deadlock. For example, if we have two programs that are session typed that both start with a receive, then we would be able to show these programs are not dual and can therefore reject the programs. On the other hand if we do have programs that are dual then we can be certain they will not deadlock and will not become desynchronized. The dual of a session type can be computed using a type family.

```
type family Dual (a :: ST) where
  Dual (a :!> r) = a :?> Dual r
  Dual (a :?> r) = a :!> Dual r
  Dual Eps = Eps
```

We already said the dual of `:!>` is `:?>`, but that also means that the dual of `:?>` is `:!>`. For some session types the dual is itself. The `Eps` session type represents an empty session type and the dual of that can only be the empty session type. We will use duality to reject session typed programs that are not dual. In chapter 4 we will use `Dual` to implement this. For now we will expand `Dual` each time a new session type is introduced.

3.1.2 DSL

We use session types to give our programs a static guarantee at compile time that our program does not deadlock or become unsynchronized. We will somehow have to denote that functions implement the static guarantees that session types give us. We want to write a domain-specific language (DSL) that defines a set of primitive functions that are annotated with session types, where each primitive function somehow implements the semantics corresponding to the session type that it is tagged with. It is then possible to write a session typed program by composing these primitive functions. There are actually two types of DSLs [14]: Shallow embedding and a deep embedding. In this section we will discuss both and compare their merits.

A shallow embedded DSL consists of a set of primitive functions that directly implement the semantics of the DSL. For Cloud Haskell semantics that means we call a Cloud Haskell function, within the definition of the primitive, that corresponds to the session type of that primitive. We define primitives for each session type.

```
newtype Session s a = Session { runSession :: ProcessId -> Process a }
send :: Serializable a => a -> Session (a :!> r) ()
send a = Session $ \pid -> CH.send pid a
recv :: Serializable a => Session (a :?> r) a
recv = Session $ \_ -> CH.expect
eps :: a -> Session Eps a
eps a = Session $ \_ -> return a
```

Each function returns a data type `Session` that has the session type as a phantom type and carries a `ProcessId` of the process that the current process is in session with. The function definitions construct this data type and call the Cloud Haskell function that corresponds to the session type. For `:!>` this would be `send` and for `:?>` this is `expect`. The main advantage of using a shallow embedding for DSL is in its simplicity. The functions are essentially wrappers around a piece of Cloud Haskell code that we annotated with an extra type. However, when using a shallow embedding we are committing to a single semantics, in this case the Cloud Haskell semantics.

A deep embedding decouples evaluation of a DSL from the DSL API by implementing the semantics in some interpreter function that evaluates an abstract syntax tree (AST) generated by the functions defined in the API. To create a deep embedded DSL for session types we first need to describe terms that compose the AST. We do this by defining a GADT named `STTerm` whose constructors', the terms, are annotated with session types.

```
data STTerm :: ST -> Type -> Type where
  Send :: (a :: Type) -> STTerm r b -> STTerm (a :!> r) b
  Recv :: (a -> STTerm r b) -> STTerm (a :?> r) b
  End :: a -> STTerm Eps a
```

The structure of each constructor reflects the structure of the session type that it is annotated with. For example, the session type `:!>` takes an argument `a` and another argument `r`. We have that `Send` also takes this argument `a` and another `STTerm` that is annotated with `r`. This is similarly so for `:?>`, except for that we cannot give `Recv` an argument `a` until evaluation. Instead we give `Recv` a continuation that given a value of type `a` produces the `STTerm` that is annotated with the continuation of the receive session type. The third constructor `End` does not take a `STTerm` as an argument, since `Eps` does neither. We do give it a value as an argument that can be used as the result of the protocol. We will one again show an example of the ping pong protocol, but this time using the deep embedding.

```
prog2 :: STTerm (Ping :!> Pong :?> Eps) Pong
prog2 = Send Ping $ Recv $ \pong -> End pong
```

This example also shows why it is necessary to have a continuation in `Recv`. At the time of writing a session typed program we don't have access to the value that will eventually be received. This is only given to us during evaluation.

The second part of our deep embedding is defining an interpreter that takes a program like `prog2` as an argument and implements the desired semantics, which in this case are the Cloud Haskell semantics. We define a simple interpreter named `eval` that takes a `STTerm` and a `ProcessId` and returns a `Process a`.

```
eval :: STTerm s a -> ProcessId -> Process a
eval (Send a r) pid = CH.send pid a >> eval pid r
eval (Recv r) pid = CH.expect >>= eval pid . r
eval (End a) _ = return a
```

The implemented semantics are no different from the shallow embedding. To fully evaluate an abstract syntax tree we have to traverse it and evaluate each node in the tree, but this is easily done by simply recursively calling `eval` on the `STTerm` argument of each constructor (if there is one).

As said before, the biggest advantage of a deep embedded DSl over a shallow embedded DSL is that we can define different interpreters that each implement different semantics without having to change our API. This means that we can take a piece of code and simply pass it to different interpreters to get different types of results. This is only possible with a shallow embedding if you defined functions for different semantics. We have chosen to use a deep embedding for our DSL for this specific reason as it can make our library much more powerful than it would otherwise be. In later chapters we also show different interpretations that we could think of and that are useful. Of course there is also a downside to using a deep embedding, besides being more complex, and that is mostly performance related. Compared to the shallow embedding we have to generate an AST representing a program and once generated we evaluate it. This is much more work then directly evaluating it. In chapter 7 we perform benchmarking tests to see exactly how big this penalty is.

3.2 Composing session type programs

To write a session typed program using our deep embedded DSL you have to make use of the constructors of `STTerm`. You combine them in such a way that it represents a program with the desired semantics. Often though you would like to split up your code in different functions. The larger a function becomes the more unclear it becomes what each part of the function is actually doing, furthermore it becomes more difficult to generalize certain computations. This can also be the case for session typed programs and we should therefore also be able to compose separate session typed programs to form one larger session typed program. The current way to do this is by having a function take an argument that is placed as the continuation of the session. To illustrate what we mean by this we split the ping pong example in two different functions that we then compose.

```
sndRecv r = Send Ping $ Recv \pong -> r pong
end a = End a
prog = sndRecv end
```

This form of composition is not the most user friendly. First of all it requires that if you want to make a function composable you have to explicitly give it a parameter. Furthermore in section 3.4, we introduce a form of branching to session types, which means that there should be a parameter to the function for each branch that is composable. Also, arguments given to `sndRecv` must be functions that take a value that is eventually returned by the receive. In this example we happen to have exactly such a function, but not is not necessarily the case. And in that case you would have to explicitly ignore the value returned by the receive using a lambda function that ignores its argument.

3.2.1 Monads

Library writers often turn their data types into instances of `Monad`. There are several benefits to doing so. First we have that functions that return a data type that is an instance of `Monad` may be written using do-syntax. The do-syntax in Haskell refers to writing code in an imperative style, while remaining completely functional. It is often viewed that using do-syntax increases the readability of code compared to using explicit binds.
`Monad` is a type class that works as an interface for bind and `return`. However, there are many more functions that can be used on any arbitrary monad. Simply by constraining the function with `Monad` allows the function to be used by any data type is an instance of `Monad`. Meaning that a data type that is made to be an instance of `Monad` gets an extensive set of functions for free.
The third benefit of using `Monad` is that monadic code is highly composable. Essentially any two monadic functions can be composed without requiring extra parameters assuming that both functions use the same monad. We will see that making `STTerm` an instance of monad drastically improves its usability.

To make STTerm into an instance of Monad we have to give definitions for return and >>= (bind). The function return can be used to wrap a value into the data type. Since we have several constructors we need to pick one and use that. At first sight End seems most suitable, because it takes an argument that has the result type of the data type. However, we can't forget session types here. Using any of the three constructors anywhere in a program will affect the session type, which we wouldn't want to do when we use return. We therefore make a change to STTerm.

```
data STTerm :: ST -> Type -> Type where
  Send :: (a :: Type) -> STTerm r b -> STTerm (a :!> r) b
  Recv :: STTerm r b -> STTerm (a :?> r) b
  Ret :: a -> STTerm s a
```

We have replaced End with Ret whose session type is left unconstrained. It might also look like we have completely removed the session type Eps and while we have done so in the GADT, we have not in our DSL. This will be explained soon enough.

We can now try to give definitions to return and bind for STTerm.

```
instance Monad (STTerm s) where
  return a = Ret a
  (Send a r) >>= f = Send a (r >>= f)
  (Recv r) >>= f = Recv $ \a -> (r a >>= f)
  Ret a >>= f = f a
```

The above definition is actually not type correct, but before we explain why, we will first explain the definitions and describe the API.
The function return simply wraps a value into Ret and for its definition in bind we can drop the constructor and simply apply the composing function to the value wrapped in Ret. We also give a definition of bind for Send and Recv. All we do there is recursively compose the continuation of a session type term with the given function. What we also realize here is that STTerm has a free-like structure [20]; we have a constructor representing a return and several constructors that do some kind of unrolling of its recursive argument. We will see in later chapters why realizing this is important.

We would now like to redefine our example that composed two simple session type programs using binds and possibly returns. However, the first argument of bind may not be a function. This means we cannot use Send a as the first argument to a bind nor can we use the function sndRecv in the same way as it still requires an argument for composition. To solve this we write wrapper functions around each constructor such that they do not require an extra argument for composition, but can still be composed using bind.

```
send :: a -> STTerm (a :!> r) ()
send a = Send a $ Ret ()
recv :: STTerm (a :?> r) a
recv = Recv Ret
```

```
end :: a -> STTerm Eps a
end = Ret
```

For `send` and `recv` we use a `Ret` as its continuation knowing that if it is used in a bind it will be replaced. By passing `Ret` to `Send` and `Recv` we ensure that they are fully applied and are of type `STTerm s a`, allowing them to be composed using bind. We also introduce a wrapper for `Ret` itself, but now it is annotated with `Eps`.

We can now revise the example program to the following:

```
sndRecv = send Ping >> recv
prog = sndRecv >>= end
```

Unfortunately, our current implementation is actually not type correct. Namely, the definition of bind for both `Send` and `Recv` do not type check. Bind has the following type `m a -> (a -> m b) -> m b`. Let's try replacing `m` with the type of `Send`.

```
STTerm (a1 :!> r) a -> (a - STTerm (a1 :!> r) b) -> STTerm (a1 :!> r) b
```

That means that `f`, the second argument in our bind definition, has type

```
a -> STTerm (a1 :!> r) b
```

However, we must also compose this function with the continuation of `Send`, which has type `STTerm r a`. It seems that we cannot compose GADTs with differently tagged constructors using the normal monadic bind. To solve this problem we will make `STTerm` into an indexed monad [10].

3.2.2 Indexed monads

Indexed monads are monads that, besides a result type, have two additional type parameters that denote the state of the monad. The first being the initial state and second the final state. We define the interface for indexed monads as follows:

```
class IxMonad (m :: p -> p -> Type -> Type) where
  (>>=) :: m s t a -> (a -> m t k b) -> m s k b
  (>>) :: m s t a -> m t k b -> m s k b
  return :: a -> m i i a
  im1 >> im2 = im1 >>= \_ -> im2
```

The interface is very similar to that of the `Monad` type class, except for that `m` now takes two additional type parameters of the same kind. Furthermore, to bind two indexed monadic computations it is necessary that the final state of the first computation equals the initial state of the second computation. The resulting computation's state is then equal to the initial state of the first computation and its final state is equal to that of the second computation. A `return` does not modify state as it only wraps a value into the indexed monad.
We once again need to modify `STTerm` such that we can make it an instance of `IxMonad`.

```
data STTerm :: ST -> ST -> Type -> Type where
  Send :: (a :: Type) -> STTerm r r' b -> STTerm (a :!> r) r' b
  Recv :: (a -> STTerm r r' b) -> STTerm (a :?> r) r' b
  Ret :: a -> STTerm s s a
```

The initial state of `Send` is simply `a :!> r`. It then passes `r` as the initial state of its continuation. The continuation produces a final state, which is then also the final state of `Send`. This works similarly for `Recv`. Also like `return`, `Ret` does not modify state.

We have now made the necessary modifications to make `STTerm` an instance of `IxMonad`.

```
instance IxMonad STTerm where
  return = Ret
  (Send a r) >>= f = Send a (r >>= f)
  (Recv r) >>= f = Recv (r >>= f)
  (Ret a) >>= f = f a
```

These definitions are no different from the definitions in the `Monad` instance, however the type of the indexed bind ensures that this definition does type check.

To accommodate for the extra type parameter given to `STTerm` we also have to modify the wrapper functions.

```
send :: a -> STTerm (a :!> r) r ()
send a = Send a $ Ret ()
recv :: STTerm (a :?> r) r a
recv = Recv Ret
end :: a -> STTerm Eps Eps a
end = Ret
```

For both `send` and `recv` the continuation session type is simply `s`. For `end` it stays `Eps`, since we cannot change a session type after it has been emptied and neither would the type of `Ret` allow us to type it any differently.

As it is we still cannot use do-syntax for our DSL. Using do-syntax translates to a sequence of monadic binds. However, it uses by default the bind of `Monad`. We can change this by enabling the `RebindableSyntax` extension. Then do-syntax will translate to a sequence of `(>>=)` that is in scope. Hence, then all we need to do is ensure that the bind of `IxMonad` is in scope and that of `Monad` is not. A clear downside of this is that you cannot mix do-syntax for indexed monadic code with that of monadic code. To close this section we give the ping pong example that is written using do-syntax.

```
prog :: STTerm (Ping :!> Pong :?> Eps) Eps ()
prog = do
  send Ping
  Pong <- recv
  end ()
```

3.3 Interleaving computations

So far we've only considered how to construct a session type program that can send and receive messages and besides regular pure computations we can't do much else. For example, we might want to print a received message to console, or perhaps the message contains the name of a file that we need to read. Both of these actions require that we can do `IO`. To facilitate this we make `STTerm` into a monad transformer such that we can lift monadic computations. First we'll have to define an indexed monad transformer type class.

```
class IxMonad (t m) => IxMonadT t m where
  lift :: m a -> t m s s a
```

We also have to add a new constructor to `STTerm` that stores monadic computations.

```
data STTerm :: (Type -> Type) -> ST -> ST -> Type -> Type where
  ..
  Lift :: (m (STTerm m s r a) :: Type) -> STTerm m s r a
instance M.Monad m => IxMonad (STTerm m) where
  ..
  (Lift m) >>= f = Lift $ m M.>>= \st -> M.return st >>= f
instance Monad m => IxMonadT STTerm where
  lift m = Lift $ m >>= return . Ret
```

`STTerm` gets yet another type parameter that denotes the type of the underlying monad. `Lift` is given a monadic computation that returns `STTerm`. In the bind definition we first extract a value from `m`, which we can then bind with `f`. Note that two different instances of bind are used there. The function `lift` runs the computation `m` and stores the result in an `Ret`. Being able to lift computations is a powerful tool to have. A simple example can be seen below.

```
io :: MonadIO m => IO a -> STTerm m s s a
io = lift . liftIO
prog :: STTerm m (a :?> Eps) Eps ()
prog = recv $ \x -> io (putStrLn x) >> end ()
```

3.4 Branching

Using `:!>`, `:?>` and `Eps` we can write session type programs that are of a linear structure, but often do we want to make a choice in the protocol depending on some value. For example, we might want to write a program like this:

```
client :: String -> ByteString ->
          STTerm m (String :!> Either String ByteString :?> ..) Eps ()
client fname newdata = do
  send fname
```

```
  eth <- recv
  case eth of
    Left err -> io (putStrLn err) >> end ()
    Right _ -> send newdata >> end ()
```

We have a client that communicates with a server storing file data. The client wants to modify the data of a file. It does so by sending the server the file name, after which the server sends back the file data to the client. Since the file name passed to the server might not correspond to an existing file, the server returns an `Either String ByteString`. If the file did not exist, the server returns `Left` together with an error in the form of a string, and otherwise it returns `Right` together with the file data in the form of a `ByteString`. The client has to make a choice depending on what it receives. Usually, you would make a choice using an if-or-case expression as is done in the example. If the client received an error, then it simply prints it to console and subsequently ends the protocol. However, if the client did receive the file data, then it knowns the file existed and replaces it with new data. The difference in the two choices is that the second choice requires an extra send. We have to remember that we want to do this session typed, but that poses a problem. In the type signature of `client` I had already left out a part of the session type for good reason. We have to express the session type that comes after the `recv`, but we cannot choose a specific branch to form this session type. We have to somehow encode the branching in the session type as well, such that we can describe the session types of both branches.

For this purpose we extend `ST` with two more session type operators that encode branching.

```
data ST = .. | (:+:) ST ST | (:&:) ST ST
```

We have introduced two new session types: `:+:` represents selecting a session type branch and `:&:` represents offering a choice between two session type branches. With selecting a branch we mean that following the selection the program implementing this session type must adhere to the session type in the selected branch. An offering means that a program must prepare code that is specific to each branch and have it be prepared to execute either branch. We can illustrate this with the example in the beginning of this section. The client receives an `Either String ByteString`. If the contained value is a `String`, then the client chooses the left branch and if it is a `ByteString` it chooses the right branch. The client is actively selecting a branch and requires `:+:` to represent this selection. Dually the server will have to expect that the client selected either one of the branches. It is in fact offering the client to select a branch and it represents this offering of branches using `:&:`. Using these two session type constructors we can now fully write the type signature of `client`:

```
client :: String -> ByteString ->
          STTerm (String :!> Either String ByteString :?>
                  (Eps :+: ByteString :!> Eps))
```

The session type operators for send and receive have a higher priority than that of the branching session types. Hence why `Eps :+: ByteString :!> Eps` is parsed like `Eps :+: (ByteString :!> Eps)`. Next we will add new constructors to `STTerm` so that we have a way of representing branching at term level.

```
data STTerm :: (Type -> Type) -> ST -> ST -> Type -> Type where
  ..
  Sel1 :: STTerm m s t a -> STTerm m (s :+: r) t
  Sel2 :: STTerm m r t a -> STTerm m (s :+: r) t
  Offer :: STTerm m s t a -> STTerm m r t a -> STTerm m (s :&: r) t a
```

We introduce three new constructors to `STTerm` to support branching. For making a choice we add `Sel1` and `Sel2`. Both constructors only take one continuation `STTerm`, which is encoded with the session type of the branch it represents. `Sel1` representing the left branch and `Sel2` the right branch. `Offer` takes two possible continuation `STTerms`, one for each possible branch. The bind definition for all three constructors are fairly straightforward, except that for `Offer` we compose bind `f` with both `s` and `r`, since any computations after an `Offer` have to consider that either branch was taken and should therefore be composable with both.

```
(Sel1 r) >>= f = Sel1 $ r >>= f
(Sel2 r) >>= f = Sel2 $ r >>= f
(Offer s r) >>= f = Offer (s >>= f) (r >>= f)
```

We also have to change the `Dual` type family.

```
type family Dual (s :: ST) where
  ..
  Dual (s :+: r) = Dual s :&: Dual r
  Dual (s :&: r) = Dual s :+: Dual r
```

It should be of no surprise that the dual of making a choice is offering a choice, and the dual of offering a choice is making one.

Evaluating the branching constructors comes down to choosing a branch and afterwards evaluating that branch, while ignoring the other.

```
eval :: ProcessId -> STTerm Process s r a -> Process a
eval pid (Sel1 r) = send pid True >> eval pid r
eval pid (Sel2 r) = send pid False >> eval pid r
eval pid (Offer s r) = do
  b <- expect :: Process Bool
  if b then eval pid s
       else eval pid r
```

We choose a branch by sending a `Bool` to the other process. That process then knows which branch we took by checking the value of the boolean. We use `True` for choosing

the left branch, and `False` for choosing the right branch.

As a final touch to this section we add wrapper functions for branching and use those to give a correct implemention of `client`.

```
sel1 :: STTerm m s t a -> STTerm m (s :+: r) t a
sel2 :: STTerm m r t a -> STTerm m (s :+: r) t a

client fname newdata = do
  send fname
  eth <- recv
  case eth of
    Left err -> sel1 >> io (putStrLn err) >> end ()
    Right _ -> sel2 >> send newdata >> end ()
```

This time we use `sel1` and `sel2` to select a branch, after which session type operations are used that are specific to the selected branch.

3.5 Recursion

Often, one wants to repeat a certain action several or even an unknown number of times. In Haskell this is generally implemented using recursion. Unfortunately, we are unable to repeat session typed actions a potentially infinite number of times as it is. Take the following program for example:

```
sendChar = send 'c' >> sendChar
```

All this program does is send the character 'c', but the program doesn't even type check giving us the following error message:

```
Occurs check: cannot construct the infinite type: r ~ Char :!> r
```

Apparently GHC tries to construct a session type that has an infinite occurrence. To understand why this error is generated, let us first recall the session type for sending: `(a :!> r)`. Since we are sending a character, we have that `a` unifies with `Char` giving us the type `Char :!> r`. The final state of `send 'c'` remains `r` and must be equal to the initial state of `sendChar` as is required by the use of the indexed monad bind. The definition of `sendChar` starts with `send 'c'`, which means the initial state of `sendChar` must be equal to `send 'c'`. We now have that the initial and final state of `send 'c'` must be equal to the initial state of `sendChar`. However, that would only be possible if the initial and finale state of `send 'c'` are the same, but this is not the case. The produced type error essentially explains that the initial state of `sendChar` must be equal to both `r` and `Char :!> r`.

We will need some additional type machinery to be able to write recursive session type programs. We add three more session type constructors to `ST`:

```
data ST = .. | R ST | V | Wk ST
```

The first session type `R` delimits the scope of recursion. Within that scope we use a recursion variabled, denoted by `V`, to mean the actual point of recursion with the idea that, at term level after the use of `V`, all session type operations from the beginning of the scope that `R` delimits have to be repeated. For instance the following session type: `R (Char :!> V)` says that after enabling recursion we do a send. The recursion variable that comes after the send tells us that we then have to repeat the send.

Occasionally we need nested recursion. Since the recursion variable recurses to the most inner `R`, we need another session type that allows us to broaden the scope of recursion. We use the `Wk` session type for this purpose.

```
R (Int :!> R (Wk V :+: Char :?> V))
```

The example session type is a bit more involved, since to show the use of `Wk` we need to enable recursion twice and we need branching if we want to use two recursion variables. Using `Wk` right before the recursion variable will weaken the scope delimited by the most outer `R`. Therefore the next action to be done is sending an `Int`. In the second branch we first do a receive followed by recursing back to the most inner `R`.

We need a way to enforce that the session typed operations occuring after the recursion variable have the same session type as the argument session type of the `R` that enabled this recursion. We can define a term that replaces `V` with the appropriate session type, but in doing so we need know what this session type is. In the work done by Pucella and Tov they use a data type called a capability that holds a context, a list of types, and a session type. Instead of tagging a term with a session type they tag it with that data type. We will do the same.

```
data Cap = Cap [ST] ST
```

We change `STTerm` such that it is now indexed by `Cap` and we add three constructors for recursive session types.

```
data STTerm :: (Type -> Type) -> Cap -> Cap -> Type -> Type where
  ..
  Rec :: STTerm m (Cap (s ': ctx) s) r a -> STTerm m (Cap ctx (R s)) r a
  Var :: STTerm m (Cap (s ': ctx) s) r a -> STTerm m (Cap (s ': ctx) V) r a
  Weaken :: STTerm m (Cap ctx s) r a -> STTerm m (Cap (t ': ctx) (Wk s)) r a
```

The context in `Cap` is being used as if it were a stack, where the top most session type points represents the current scope delimited by the most inner `R`. The constructor `Rec` further delimits the scope by pushing the session type argument `R` on the context. Since subsequent `R's` always have increasingly smaller session types, the scope will always decrease when pushing a session type on the context. For the opposite effect we can pop session types from the context to expand the scope as is done in the type of `Weaken`. The idea that the scope is defined by the session type at the top of the context is validated by the `Var` constructor that takes the session type on top of the context as the initial

state of its argument `STTerm`. Afterall recursion always recurses back to the beginning of the current scope.

Let us now try to these new constructs to solve the earlier mentioned type error. First we define wrapper functions for our session type terms:

```
recurse :: STTerm m (Cap (s ': ctx) s) r a -> STTerm m (Cap ctx (R s)) r a
var :: STTerm m (Cap (s ': ctx) s) r a -> STTerm m (Cap (s ': ctx) V) r a
wk :: STTerm m (Cap ctx s) r a -> STTerm m (Cap (t ': ctx) (Wk s)) r a
```

Unlike previously defined session type primitives, we now ask continuations to be given to these functions. We do this, because we believe that a user would prefer to apply `recurse` or `var` to an argument as these are usually used at the start and end of a function. To give the user more options we also provide functions that take no argument and have to be composed using bind: `recurse0`, `var0` and `wk0`. We can now rewrite `sendChar` as follows:

```
client :: String -> ByteString ->
          STTerm m (Cap ctx (R (String :!> Either String ByteString :?>
                               (V :+: ByteString :!> Wk Eps)))) r ()
client = recurse . client'
client' fname newdata = do
  send fname
  eth <- recv
  case eth of
    Left err -> do
      sel1
      io (putStrLn err)
      fname' <- io readLine
      var $ client' fname' newdata
    Right _ -> do
      sel2
      send newdata
      wk0
      end ()
```

We once again use the client program from the previous section. With the addition of recursion, we can now make it more complete. We start by enabling the recursion using `recurse`. Then in the first branch, where the server was not able to find a file with the given file name, we can now request a new file name and repeat a part of the protocol. In the second branch we do a `wk0` to empty the context. This isn't strictly necessary since we end the protocol right after that, but it is good practice to expand the context after it had been delimited and recursion within that scope no longer occurs. This way you can avoid accidental recursion when you don't delimit the scope where you want to delimit it.

The dual of `R`, `Wk` and `V` are themselves. Whenever a process recurses, we require that the other process in the session does the same. Otherwise, it can become easy for a process to repeat actions more often than the other process. We also do not change `Dual` to accept a `Cap` instead of a `ST`. We will assume that a session typed program always starts with an empty context. Then it is not necessary to also dualize the contents of a context. The definitions for bind and `eval` are entirely straightforward. For bind we simply apply `f` to the argument of the constructor and in `eval` we only need to evaluate the argument.

```
Dual (R s) = R (Dual s)
Dual (Wk s) = Wk (Dual s)
Dual V = V
```

```
eval (Rec s) pid = eval s pid
eval (Weaken s) pid = eval s pid
eval (Var s) pid = eval s pid
```

In this chapter we have introduced a DSL for writing session typed programs. Furthermore, we also described a way to make a translation from the semantics of a session typed program to those of the Cloud Haskell library. But we are still missing some mechanisms that allow us to actually run and evaluate sessions from within a Cloud Haskell program. In the next chapter we will introduce these mechanisms and solve some remaining problems after which we have a fully functional library.

4 Connecting to Cloud Haskell

The library that we are writing should be usable in combination with the Cloud Haskell library. Specifically, a user should be able to spawn a session, two conversing session typed programs, from within a Cloud Haskell program. Besides that, basic primitives have made up our example session typed programs so far, but you should be able to use all Cloud Haskell functions from within a session typed program. You can simply lift most of these functions, but some of these functions require some extra effort to be usable. We've mostly ignored the `Serializable` constraint. We have to use it to constrain `eval`, where we make use of `send` and `recv`. However, the types to be constrained are hidden in the session type, which in the type of `eval` is polymorphic. The final problem we will be addressing in this chapter is the potential that a session becomes unsynchronized, which may occur when the administrative communication required for branching mixes with actual values sent or received in the program.

4.1 Spawning sessions

The mechanism for spawning a `Process` is to use `spawnLocal` and `spawn`, which spawn a new `Process` on the local node or a remote node respectively. We want to do something similar for spawning sessions.
The first combinator we write is `spawnSession` that spawns a session on the local node.

```
type Session s r a = STTerm Process s r a
spawnSession :: Session s r a -> Session (Dual s) r a -> Process a
spawnSession ss1 ss2 = do
  mpid <- getSelfPid
  othpid <- spawnLocal $ eval ss2 mpid
  eval ss1 othpid
```

The function takes two session typed programs as arguments. It spawns a process to evaluate the second program and evaluates the first program on the current process. For evaluation a process needs to know the `ProcessId` of the other process in the session. This is given to the first use of `eval` by taking the `ProcessId` of the current process. The result of spawning a process is the `ProcessId` corresponding to the spawned process. This can then be passed to the second use of `eval` together with the remaining program. In this function we also see the first use of `Dual`. To enforce that two session typed programs can actually be used to form a session we use `Dual` applied to `s` to say that the second program must implement the dual session type of the first program, thereby ensuring that these two programs can together correctly implement the given protocol.

Something to note here is that we are using `spawnLocal` for the first session typed program and do not spawn a new process to compute the second session typed program. Unlike `spawnLocal` and `spawn` we have to spawn two processes to run a session, but this also gives us more options. We can decide to simply spawn two new processes or spawn a single process as we have done in `spawnSession`. The difference being that `spawnSession` is now a blocking function, whereas a function that spawns two processes is not. We also have more options regarding where to spawn these processes. For example, we could spawn one process locally and the other remotely. Overall there are six different ways to spawn a session.

```
asyncLocalSession :: Session s r a ->
                     Session (Dual s) r a ->
                     Process (ProcessId, ProcessId)
asyncSession :: NodeId ->
                Session s r a ->
                Session (Dual s) r a ->
                Process (ProcessId, ProcessId)
```

Above we have given two more functions for spawning sessions. Both spawn two processes that are both spawned locally or both spawned remotely and as such are non-blocking. For all non-blocking functions that spawn sessions we can also return the `ProcessIds` of the spawned processes. We do this to allow untyped messaging from within the session itself.

Finally we introduce one more type of function for spawning sessions.

```
spawnSessionS :: Session s r a -> Session (Dual s) r a -> Session t t a
spawnSessionS = lift . spawnSession
```

This function has the same semantics as `spawnSession`. The only difference being that this function can be directly used within `Session`. It should be noted that the session types of the arguments passed to `spawnSessionS` are completely independent from the session type of the current session. In other words, spawning a session from within a session should not change the state of the session. This is reflected in the session by having the initial state and final state of the session be the same. Our session type solution cannot handle multi-party sessions [8], where a protocol is executed by more than two processes, but with `spawnSessionS` we can still allow a running sessions during a session without affecting the protocol of the session.

Using one of the above mentioned spawn functions we can finally run a session. For example, we can take the client program from section 3.4, construct the server program as well and run the session using `spawnSession`.

```
client fname newdata = do
  send fname
  _ <- recv
  sel2
  send newdata
  end ()
```

```
server = do
  fname <- recv
  data <- io $ IO.readFile fname
  send data
  offer (end ()) $ do
    newdata <- recv
    io $ IO.writeFile fname newdata
    end ()
```

We have slightly modified the protocol. We will now assume that a file with the given file name exists on the server and the client can now choose to update or not to update the file using a selection operator. In this case it chooses to update the file using `sel2`. To support this selection, the server has to offer both both options. In the first branch it simply ends the session and in the second branch does it do another receive before ending the session. With `spawnSession` we now have the ability to run this session: `spawnSession client server`, which is then evaluated to `Process ()`. However, the definition of `Dual` is somewhat faulty, as will be explained in the next paragraph, and in this case will throw a type error.

We have on purpose left out the type signatures of both `client` and `server`. While it is recommended to always give a type signature to functions for improving readability, sometimes you want to make use of type inference to figure out the type for you. Especially, if the type can become somewhat complex. Type inference computes the following two types for the functions:

```
client :: Session (String :!> ByteString :?> (s :+: ByteString :!> Eps))
server :: Session (String :?> ByteString :!> (Eps :&: ByteString :?> Eps))
```

We had to leave the first branch in `client` unconstrained as we do not define it at term level in `client`. Therefore type inference cannot figure out the type for this branch. The function `spawnSession` requires that the dual session type of `client` must be equal to the session type of `server`. However, `Dual` is unable to compute what the dual session type of the first branch should be as it was left unconstrained. The type family `Dual` becomes stuck and cannot fully compute the dual session type with the following session type as a result.

```
String :?> ByteString :!> (Dual s :&: ByteString :?> Eps)
```

The computed session type is clearly not equal to the session type of `server` resulting in a type error. To solve this we can add a type family dependency [18] that makes type families injective.

```
type family Dual s = r | r -> s
```

By making `Dual` injective, the type checker is able to determine the arguments of a pattern if the result of the type family is known. In our case we know the result, because the result must be equal to the session type of `server`. Specifically, the first branch of

`server` has session type `Eps`. The type family can then determine that `s` in the session type of `client` must be `Eps` as well. `Dual` is now able to fully compute and for this example will see that `server` is in fact the dual of `client`.

4.2 Constraints

As we have already explained in chapter 2, to send or receive a value using the Cloud Haskell primitives it is necessary values can be serialized and deserialized. The type class `Serializable` provides us with a means to do so. The types in the Cloud Haskell functions `send` and `expect` are constrained with `Serializable`. Any value passed as an argument to `send` or is returned by `expect` must therefore be an instance of `Serializable`. We happen to use these functions in our `eval` function, but nowhere have we been constraining the types with `Serializable`. Since we do not know which types of values will be transmitted, we must also constrain these types in `eval` with `Serializable` in some way.

In the type of `eval` we do not have direct access to the types that we want to constrain. We use `eval` to pattern match on a GADT whose constructors vary in their types, which is why the session type must be left polymorphic in the type of `eval`. This problem is actually quite similar to the constrained-monad problem [17]. The constrained-monad problem explains that there are many different data types that are monad-like, but cannot be made instances of `Monad`, because of extra constraints on their operations. A solution to this problem that works for a deep embedding is to embed the constraint in the data type itself. For `STTerm` that would look something like this:

```
Send :: c a => (a :: Type) -> STTerm c m (Cap ctx r) r' b ->
                              STTerm c m (Cap ctx (a :!> r)) r' b
```

We add an extra type parameter to `STTerm` that is a constraint that we apply to all types of kind `Type` in the session type.

The above given solution works, but it is not perfect. Namely, we have to add yet another type parameter to `STTerm`, which only increases the complexity of the data type. There exists a much better solution that is possible because we have session types. In `eval` we do not have direct access to the types that we wish to constrain, but we do have access to the session type that contains those types. In de Vries and Löh *True sums of products* [4] they introduce a type family that applies a constraint to each type in a list of types. The result is a constraint set that can be used to constrain a function that constrains over all the types in the given list of types. Since session types have a very inductive structure we can write a similar type family to traverse the session type, apply a given constraint wherever possible and construct a constraint set that we can constrain `eval` with.

```
type family HasConstraint c s :: Constraint where
  HasConstraint c (Cap ctx s) = (HasConstraintCtx c ctx, HasConstraintST c s)
```

```
type family HasConstraintCtx c ctx :: Constraint where
  HasConstraintCtx c '[] = ()
  HasConstraintCtx c (s ': ctx) = (HasConstraintST c s, HasConstraintCtx c ctx)

type family HasConstraintST c s :: Constraint where
  HasConstraintST c (a :!> r) = (c a, HasConstraintST c r)
  HasConstraintST c (a :?> r) = (c a, HasConstraintST c r)
  HasConstraintST c (s :&: r) = (HasConstraintST c s, HasConstraintST c r)
  HasConstraintST c (s :+: r) = (HasConstraintST c s, HasConstraintST c r)
  HasConstraintST c (R s) = HasConstraintST c s
  HasConstraintST c (Wk s) = HasConstraintST c s
  HasConstraintST c s = ()
```

We need three type families to fulfill our goal. The type family `HasConstraint` applies a given constraint to types of kind `Type` in the session type. Often do we actually only have access to the capability of `STTerm`, so the type family takes a capability as an argument. Then we apply a different type family to the context of the capability and another one that is applied to the session type. Since the context may contain session types, for example when evaluating a `Rec`, we have to be sure that we also constrain those types. The type family `HasConstraintCtx` takes a constraint and a context. It pattern matches on the context and applies `HasConstraintST` to the contents of the context. The type family `HasConstraintST` constructs the actual constraint set. For the session types `:!>` and `:?>` it applies the given constraint to their first argument and then makes a recursive call on their second argument, thereby constraining the entire session type. For the branching session types, we actually make two recursive calls for each branch. The results are then unified into a single constraint set. For the other session types we either make a recursive call using their session type argument or we return the empty constraint set.

Using these type families we can now constrain `eval` with `Serializable`.

```
eval :: HasConstraint Serializable s => Session s r a -> ProcessId -> Process a
```

Sometimes, we might also need more than one constraint over a session type. We can reuse `HasConstraint` to apply constraints from a list of constraints to a session type.

```
type family HasConstraints cs s :: Constraint where
  HasConstraints '[] s = ()
  HasConstraints (c ': cs) s = (HasConstraint c s, HasConstraints cs s)
```

The given solution for constraining types of kind `Type` in a session type is not only polymorphic in which constraints we want to apply, but it is also polymorphic in the order that the constraints are given to `HasConstraints`. The previously mentioned solution is not polymorphic in the order of the constraints. For instance, a trivial program of the following type

```
STTerm Process '[Serializable, Show] s s ()
```

can only be passed to functions that take an argument of that exact type.

```
f :: STTerm Process '[Serializable] s s () -> Process ()
```

The function `f` would reject the above program, because the type level lists containing the constraints do not match. With our type family solution for constraining functions we do not have this problem.

In further chapters we will more often have to write type families that traverse a session type. In that case, just like `HasConstraint`, we will have exactly three type families that work on a capability, context and session type respectively. The structure of the type families applied to the capability and context will always be (almost) the same, which is why we won't define these anymore and simply assume their existence.

4.3 Intermediate evaluations

The Cloud Haskell library features an extensive API consisting of functions shown in section 2 and many more. As we're writing a library that builds on top of Cloud Haskell, it is good practice to write wrapper functions for all these functions. We simply take a function from the Cloud Haskell library and lift it to a function whose result type is of the form `Session s s`. For example, there exists a function named `getSelfPid`, which returns the `ProcessId` of the current process. For this function all we need to do is apply lift to it so that we get a function of type `Session s s ProcessId`.

```
getSelfPid :: Session s s ProcessId
getSelfPid = lift getSelfPid
```

Unfortunately, it is not always this easy. The `Process` monad has an instance of `MonadThrow`, `MonadCatch` and `MonadMask`. These three type classes together allow the use of `throw`, `catch` and `mask` within the `Process` monad. We also want to allow these functions to be used in `Session`. Defining an instance for `MonadThrow` poses no problem. We can simply lift `throw`. It is not as easy for the other two functions. First we need to consider whether `catch` (and `mask`) should be aware of the progression of a session. In other words, should we give `catch` the type

```
catch :: Exception e => Session s r a -> (e -> Session s r a) -> Session s r a
```

or the type

```
catch :: Exception e => Session s s a -> (e -> Session s s a) -> Session s s a
```

If we make `catch` aware of the progression then we would allow catching exceptions that occur from session typed operations. It is not difficult to pass a session typed program to `catch`, have that argument be evaluated and pass the result to `catch` that corresponds to the `Process` instance of `MonadCatch`. However, it is much more difficult, perhaps even

impossible, to define a function that can handle an exception and return a session of type `Session s r a`. Consider a small program that consists of three consecutive sends, where each `send` sends a value of a different type.

```
prog = catch (do
  send True
  send 'c'
  send 5) (\_ -> prog)
```

We try to catch any exceptions occuring from these three sends. Let's assume that the third send throws an exception. The type of `catch` tells us that the handler must redo the session. So all we can do is call `prog` once more. However, we also need to consider the other process in the session that had actually already progressed by doing two receives. That process also does not know the protocol must be restarted, since it has no way of knowing that `prog` is using `catch`. As a result the session is out of sync. Perhaps it is possible to make a `Session` always be ready to receive synchronization messages in the background, such that we can ensure a session stays synchronized no matter what happens. However, that seems fairly difficult to accomplish. Furthermore, is it even necessary to make `catch` be aware of a session's progression? Most exceptions will most likely occur when lifting `IO` computations and if an exception does interrupt the session, then we can always let the session itself crash and use `catch` of the `Process` instance to catch this exception. Hence, why we will make `catch` and `mask` oblivious to the progression of a session.

Upon defining `catch` and `mask` we encountered another problem. To define `catch` we have to lift the `catch` function that is under the instance of `Process`.

```
catch :: Exception e => Process a -> (e -> Process a) -> Process a
```

To supply arguments of the correct type we transform `Session s s a` to `Process a` and `e -> Session s s a` to `e -> Process a`. But the only way to do this is by evaluating the sessions, which requires knowing the `ProcessId` of the other process in the session. The only place where we have that knowledge is in `spawnSession`. There we could potentially pass the `ProcessId` to the given session, which should then be expecting a `ProcessId`. Then have `catch` expect the same so we can explicitly pass the `ProcessId` to `catch`. While not incorrect, it can be annoying to have to pass this `ProcessId` around explicitly. Instead we can use make a reader monad pattern [12] that has the `ProcessId` as its environment ensuring that same `ProcessId` is always used and that we can access the `ProcessId` as long as we are in a `Session`.

```
newtype Session s r a = Session { runSession :: ProcessId -> STTerm Process s r a }
```

Using a wrapper around `STTerm` does require that we also wrap all session typed functions, but these definitions are all very straightforward.

4.4 Session typed channels

In our `eval` function we use untyped messaging as our communication method. Unlike typed channels, it allows for messages of different types to be transmitted, but there exists the problem that it can lead to synchronization issues or semantical changes during runtime. For example, a process in a session might have to communicate with some outside process. It will have to do so in a non-session-typed manner, because we do not allow for multi-party sessions [8]. The outside process might send a boolean to a process in the session, but at the same time the other process in the session might also send a boolean that determines which branch it took in the protocol. If the receiving process does not receive the booleans in the same order that they were sent in, then it could mean that the receiving process will take a wrong branch to evaluate. To further illustrate this problem we show three code snippets below.

```
pOth pid = do        pA = do          pB = do
  send pid True        sel2             y <- lift (expect :: Process Bool)
  end ()               end ()           offer (send 'c' >> end ())
                                                (end ())
```

The second and third snippet show two processes that are in a session. Process `pA` selects the second branch in the protocol. Process `pB` does a non-session-typed receive of a boolean using `expect`. Afterwards it offers a choice between two branches. We also have `pOth` that is a process that is not in any session. It is passed the `ProcessId` of `pB` and uses it to send the value `True`. The `sel2` in process `pA` sends the value `False` to `pB` to select the second branch. If we have that `pB` first receives a value from `pA`, then that is the value taken by `y`. Consequently, the boolean received internally by `offer` has the value `True` and came from `pOth`. Because of that the `offer` decides to evaluate the first branch. However, `pA` clearly selected the second branch; the session has desynchronized. Instead of branching `pA` could have simply done a session typed send that sends a boolean to `pB`. That doesn't lead to synchronization issues, but it does lead to unexpected behavior of the program.

Ultimately what leads to these synchronization issues and semantical changes during runtime is our inability to separate messages that are sent or received by session typed operations from those that are sent or received by non-session typed operations. A fairly simple solution to this problem would be defining a custom data type that is annotated with the type of the value it carries and using that data type for the session typed operations. A more interesting solution involves session types and typed channels. A typed channel also gives us this separation of messages, but it currently can only send messages of a single type. In the next sections we will aim to solve this problem using session types.

4.4.1 Typed channels

The typed channel problem seems very similar to that of homogeneous lists. The list type `[a]` in Haskell is a homogeneous list. Similar to a typed channel it can only contain

values of the same type. Occasionally it is necessary to have a data structure that can contain values of different types. For that purpose you would use a heterogeneous list [11]. A heterogeneous list can be defined as follows:

```
data HList xs where
  HNil :: HList '[]
  HCons :: x -> HList xs -> HList (x ': xs)
```

A GADT is used to define a list-like structure composed of two constructors that are each tagged with a type level list. The constructor `HNil` represents the empty list and is also tagged with the empty list type. Subsequently, `HCons` represents the cons constructor that stores a value and the tail of the list. The constructor is also tagged with a list whose head is equal to the type of the stored value. The tail of the list type comes from the second argument of the constructor, which also represents the tail of the list. A heterogeneous list like `HCons True (HCons 1 HNil)` will have type `HList (Bool ': Int ': '[])`. We are guaranteed that the order in which the types appear in the type and values are stored in the term correspond. This guarantee ensures that you can safely remove a value from the list as long as you also remove the corresponding type from the list of types. Likewise you can always safely append a value as long as you also append the type of the value to the list of types.

We can apply the idea of a heterogeneous list to define a heterogeneous typed channel. If, instead of a single type, we tag a typed channel with a list of types we can safely receive and send values of different types through this typed channel:

```
SendPort (Bool ': Int ': [])
```

Although a simple list type is not powerful enough for our purposes. Since a session type may branch, the types of values that are transmitted can differ depending on the branch that was selected. A potential heterogeneous typed channel must accommodate for this possibility. Furthermore, we also require that the channel can recurse over the types for similar reasons. In fact it appears that we are better off simply tagging the typed channel with a session type instead.

4.4.2 STChan

There are at least two approaches that we could take to implement session typed typed channels. Firstly, we can modify the existing Cloud Haskell code. When a node receives a message it will inspect the message and then either adds it to the message queue of a process or inserts it into a typed channel. We could modify the code such that messages can also be inserted into a session typed channel. The downside of this approach is that it requires some hefty modifications to the Cloud Haskell library. The other approach is to define session typed channels as some kind of wrapper over typed channels. This is the approach that we have taken.

First we define a data type for messages. The content of a `Message` is existentially

quantified allowing us to insert a value of any type. The only way to then extract the type of the value is to use `unsafeCoerce`.

```
data Message = forall a. Serializable a => Message a deriving Typeable
```

Next we define wrappers for `SendPort` and `ReceivePort`.

```
data STSendPort (l :: Cap) = STSendPort (SendPort Message)
data STReceivePort (l :: Cap) = STReceivePort (ReceivePort Message)
```

Both `STSendPort` and `STReceivePort` are tagged with a phantom type that has kind `Cap`. Using `Cap` and not `ST` is necessary if we also want to allow recursion. Both `SendPort` and `ReceivePort` may only transmit messages of type `Message`, which allows us to use a typed channel underneath as we are actually only transmitting messages of the type `Message`.
We first define primitives for sending and receiving.

```
sendSTChan :: Serializable a =>
              STSendPort (Cap ctx (a :!> l)) -> a ->
              Process (STSendPort (Cap ctx l))
sendSTChan (STSendPort s) a = do
  sendChan s $ Message a
  return $ STSendPort s
recvSTChan :: Serializable a =>
              STReceivePort (Cap ctx (a :?> l)) ->
              Process (a, STReceivePort (Cap ctx l))
recvSTChan (STReceivePort r) = do
  (Message a) <- receiveChan r
  return (unsafeCoerce a, STReceivePort r)
```

In `sendSTChan` we first pattern match on the first argument so we have access to the `SendPort`. Furthermore, we wrap `a` into a `Message`, which we then, together with the send port, pass to `sendChan`. We also have to return a new send port that has `a :!>` stripped from its type. This way we ensure progression of the session type. Remember that it is important for both `STSendPort` and `STReceivePort` to stay synchronized, just like two processes are supposed to be in a session. These functions work in the `Process` monad and not the `STTerm` indexed monad. We do this because we want to use the session typed channel in the `eval` function, where we are also in the `Process` monad. Furthermore, we can always lift any `Process` to a `STTerm` allowing us to also use session typed channels within sessions. The function `recvSTChan` receives a `Message`, which it unwraps to access the value within. However the type of the value is existentially quantified. This means that the type of the value could be anything and because it can be anything the type checker cannot simply give it a specific type. Fortunately, thanks to session types we do know what the type of the value is. We use `unsafeCoerce` to force the type checker to assume the type of the value is the type described in the session type.

Using these two functions we can try to partially rewrite `eval`, but as you might have already noticed there is a problem. Session types are actually too descriptive for what we need. We tag both the send port and the receive port with the same session type. This means that the send port could be forced to do a receive and the receive port a send, which isn't possible. We can use a type family to filter out all sends in the session type passed to the receive port and filter the receives in the session type passed to the send session port.

```
type family FilterSendST s where
  FilterSendST (a :!> r) = FilterSendST r
  FilterSendST (a :?> r) = a :?> FilterSendST r
type family FilterRecvST s where
  FilterRecvST (a :!> r) = a :!> FilterRecvST r
  FilterRecvST (a :?> r) = FilterRecvST r
```

Then we can rewrite `eval` to make use of our session typed channel solution.

```
eval :: HasConstraint Serializable s =>
        STTerm Process s r a -> STSendPort (FilterRecv s) ->
        STReceivePort (FilterSend s) -> Process a
eval (Send a r) sp rp = do
  sp' <- sendSTChan sp a
  eval a sp' rp
eval (Recv r) sp rp = do
  rp' <- recvSTChan rp
  eval a sp rp'
```

4.4.3 STChanT

In the revised definition of `eval` we're forced to pass the updated port from `sendSTChan` or `recvSTChan` to the recursive call of `eval`. This is necessary to progress the session type. If we have to do a sequence of sends and receives it can become especially cumbersome to explicitly pass around these ports. Of course that is not the case in `eval`, but there is no reason session typed channels may only be used within `eval`. They can even be used outside sessions. To make it more pleasant to use a session typed channel we will design an interface where we do not have to explicitly pass around the ports.

Since the ports are session typed we can, just like `STTerm`, use an indexed monad to progress the ports without having to explicitly pass them around. Different from `STTerm` is that we need to index over four type variables, two for each port. There are two ways to make an indexed monad index over four type variables. We can change `IxMonad` to take a data type that has four type parameters. If we take this approach we would have to add two type parameters to `STTerm` adding redundant type information and only making the type of `STTerm` more complex.

```
data Prod = (:*:) Cap Cap
type family Fst p where
  Fst (p :*: q) = p
type family Snd q where
  Snd (p :*: q) = q
data STChanT m (p :: Prod) (q :: Prod) a = STChanT {
  runSTChanT:: (STSendPort (Fst p), STReceivePort (Snd p)) ->
               m (a, (MTSendPort (Fst q), MTReceivePort (Snd q)))
  }
```

The other method is to define a data type `Prod` that takes two arguments. We then define another data type `STChanT` that is indexed with two type parameters of kind `Prod`. The use of `Prod` allows us to specify two types as a single type. The type class `IxMonad` only requires an initial state and final state to be equal, but does not put any constraint on how these states are defined. Therefore we can make a state as complex as we want. `STChanT` is also a state monad whose state is defined by the ports' types. These types are extracted from the type parameters of kind `Prod` using two type families `Fst` and `Snd`. We use the first argument of each `Prod` as the state of the send ports and the second argument for the state of the receive ports. As is usually the case we will start with writing wrapper functions for sending and receiving.

```
sendSTChanM :: Serializable a => a -> STChanT Process
                                        ('Cap ctx (a :!> r) :*: k)
                                        ('Cap ctx r :*: k) ()
sendSTChanM a = STChanT $ \(sp, rp) ->
                  sendSTChan a >>= \sp' -> return ((), (sp', rp))
recvSTChanM :: Serializable a => STChanT Process (s :*: 'Cap ctx (a :?> r))
                                                 (s :*: 'Cap ctx r) a
recvSTChanM = STChanT $ \(sp, rp) ->
                  recvSTChan >>= \(a, rp') -> (a, (sp, rp'))
```

We can see in both type signatures of `sendSTChanM` and `recvSTChanM` that we only need to progress one of the session types. In the definition we construct a `STChanT` to gain access to the previous state of the ports. We then pass either one of these ports to the non-M variant function that actually modifies the state of the port. Finally, we return the result of the non-M variant function together with the modified ports. The passing of updated ports is now entirely handled within these functions and henceforth don't need to be explicitly passed anymore.

Before we give another revised version of `eval` we also need to consider the other session type operations. Besides a send and receive session type, the ports can also be encoded with the remaining session types. Each that that a session typed program progresses, the ports must be progressed similarly. For example, if we select a branch in a session typed program, then we must also select the same branch in the ports. Otherwise, the port has no way of knowing which types of values are to be sent or received. Therefore,

besides for send and receive, we also define functions that modify the state of the ports for the other session types.

```
class STOperation m where
  sel1STChan :: m ('Cap ctx (s :+: r)) -> m ('Cap ctx s)
  sel2STChan :: m ('Cap ctx (s :+: r)) -> m ('Cap ctx r)
  off1STChan :: m ('Cap ctx (s :&: r)) -> m ('Cap ctx s)
  off2STChan :: m ('Cap ctx (s :&: r)) -> m ('Cap ctx r)
  recSTChan :: m ('Cap ctx (R s)) -> m ('Cap (s ': ctx) s)
  wkSTChan :: m ('Cap (t ': ctx) (Wk s)) -> m ('Cap ctx s)
  varSTChan :: m ('Cap (s ': ctx) V) -> m ('Cap (s ': ctx) s)
```

Since these operations can be used for both `MTSendPort` and `MTRecvPort` we make use of a type class. The definitions are entirely straightforward. For each of these functions we only need to update the type of the port and nothing term level needs to be done. Note that these functions also do not need to live within the `Process` monad. So all we end up doing is pattern matching and instantiating a new port. We also define M-variant for these functions.

```
sel1STChanM :: STChanT Process (Cap ctx (s :+: r) :*: Cap ctx (k :+: l))
                               (Cap ctx s :*: Cap ctx l) ()
sel1STChanM = STChanT $ \(sp, rp) -> return ((), (sel1STChan sp, sel1STChan rp)))
```

Unlike, `sendSTChan` and `recvSTChan` we now update both ports as can also be seen in the function's type. I'll omit definitions for the other functions as these follow the same pattern.

```
eval :: HasConstraint Serializable s =>
        STTerm Process s r a -> ProcessId ->
        STChanT Process (FilterRecvs s :*: FilterSends s) k a
eval (Send a r)  pid = sendSTChan a >>= eval r pid
eval (Recv r)    pid = recvSTChan >>= \a -> eval (r a) pid
eval (Sel1 s)    pid = sel1STChan >> send pid True >> eval s pid
eval (Sel2 r)    pid = sel2STCHan >> send pid False >> eval r pid
eval (Offer s r) pid = do
  b <- expect :: Process Bool
  if b then off1STChan >> eval s pid
       else off2STChan >> eval r pid
eval (Rec s)     pid = recSTChan >> eval s pid
eval (Wk s)      pid = wkSTChan >> eval s pid
eval (Var s)     pid = varSTChan >> eval s pid
```

In this definition of `eval` we still make use of untyped messaging to send and receive booleans for branching purposes. The sending and receiving of these booleans is not encoded in the session types, but we can use a type family to insert sending and receiving a boolean into the session type argument of `eval`.

```
type family SendInsBoolST s where
  ..
  SendInsIntST (s :+: r) = (Bool :!> SendInsIntST s) :+:
                           (Bool :!> SendInsIntST r)
type family RecvInsBool s where
  ..
  RecvInsIntST (s :&: r) = (Bool :?> RecvInsIntST s) :&:
                           (Bool :?> RecvInsIntST r)
```

For the send port, in the case of a selection, we want to send a boolean. Whether the first or second branch is selected does not matter. Dually for the receive port do we want to receive a boolean for an offering. In the type signature of `eval` we apply these two type families to the session types of `STChanT` and in the definition of `eval` we replace `send` and `expect` with `sendSTChan` and `recvSTChan` respectively.

The introduction of the session typed channel allows us to clearly separate communication within the session from communication that is outwards from and inwards going into the session.

4.4.4 UTChan

We need to tackle one last problem before concluding this chapter. Before we required a `ProcessId` to allow intermediate evaluation. This was mostly necessary in functions such as `catch` and `mask`. Now we'd also like for the environment in `Session` to include the send and receive port. However, `Session` is a reader monad and the type of the environment does not change. That means we can't use the session type of `Session` to determine the session types of `STSendPort` and `STReceivePort`, since the session type `Session` changes with each use of bind. Fortunately, we only need to be aware of the session types when evaluating. So instead of typed ports, we store untyped ports in the environment. An untyped port is simply a regular send or receive port that is typed with `Message`.

```
type UTChan = (SendPort Message, ReceivePort Message)
type STChan s r = (STSendPort s, STRecvPort r)
newtype Session s r a = Session {
  runSession :: ProcessId -> UTChan -> STTerm Process s r a
  }
newUTChan :: Process UTChan
toSTChan :: Proxy s -> Proxy r -> UTChan -> STChan s r
```

We also add two combinators for creating an untyped channel and one for converting an untyped channel to a session typed channel.

With the completion of this chapter we now have a fully functional session types library for writing session typed Cloud Haskell programs. In the next chapters we will

look how we can further improve the library by defining more interpreters and making optimizations.

5 Interpretations

One of the main advantages for using a deep embedded DSL over a shallow embedded DSL is that we can write different interpretation functions that apply different semantics to the same program. In this chapter we will look at several interpreters for a session typed programs.

5.1 Pure semantics

Testing code that makes use of the `Process` monad is difficult to test and debug. The `Process` monad is essentially a reader monad over the `IO` monad. A consequence is that Cloud Haskell programs are impure and allow for side effects. This should not really be a surprise as sending and receives messages over the network is an impure action, but it is exactly those actions that make these programs difficult to test. Since `Session` is a reader monad over the `Process` monad, it is no different for `Session`. Consider for example, the following program that is supposed to convert a boolean to an integer.

```
boolToInt :: Session (Cap (Bool :?> Int :!> r)) (Cap ctx r) ()
boolToInt = do
  b <- recv
  if b then send 0
       else send 1
```

On receiving `True` you would expect the value `1` to be returned and for `False` we expect `1` to be returned. The function does not correspond to this expection and we can deem it to be a bug. We would like to test this function to verify its output. We can always evaluate a `Session` from within the `Process` monad. So if we can test a Cloud Haskell program, then we can also test a session typed program. The expected way to test this function is to start a node and to run this function together with its dual as a session. The value that the dual of `boolToInt` sends can be played with to change the input to `boolToInt`. This method of testing requires the existance of a function that is the exact dual of `boolToInt`. If it does not exist then it would have to be written. Preferably, we wouldn't have to do this and instead test `boolToInt` individually. However, that can only be done if we find a different way to supply a value to `recv` that it can then return. In the next paragraph, we discuss a method on how we can accomplish exactly that.

Swierstra and Altenkirch describe in their paper *Beauty in the beast*[21] a method for purely debugging and testing impure code by defining functional and pure specifications of otherwise impure models. They begin by describing how to model teletype IO. They

define a data type IO_{tt} `a` that specifies interactions with a teletype, which together with primitive functions for each interaction describe a pure model of functions in the `IO` monad. They then define an interpreter that implements the semantics of a teletype to each corresponding interaction. The interpreter consists of three parts. The first is the first argument passed to the interpreter that are the teletype interactions. The second is a stream of input that is to be used by the interactions. The third part consist of a description of the output or behavior of the given interactions.

We will apply the method described above to purely test and debug session typed programs. IO_{tt} like monad that describes our pure model for session typed interactions. As it happens that is already our `STTerm` monad. It is entirely pure if there are no occurrences of `lift` within a program. Also, for each session typed action there exists a constructor of type `STTerm`. Next we want to describe the behavior of a session typed program. We consider three possible ways to do this. The interpreter can return an AST that describes each session typed action, the AST only describes sends and receives or the interpreter only returns the result of the program. We will first define a data type that allows us to describe the behavior of a session typed program in all three ways.

```
data Output :: Type -> Type where
  O_Send :: Show b => b -> Output a -> Output a
  O_Recv :: Show b => b -> Output a -> Output a
  O_Sel1 :: Output a -> Output a
  O_Sel2 :: Output a -> Output a
  O_Offer :: Output a -> Output a -> Output a
  O_Rec :: Output a -> Output a
  O_Var :: Output a -> Output a
  O_Weaken :: Output a -> Output a
  O_Ret :: a -> Output a
  O_Lift :: Output a -> Output a
```

The constructors are structurally the same; they have a single argument that is a continuation. `O_Send` and `O_Recv` possess an additional showable argument that is the value sent or received. Also `O_Ret` has only the return type as an argument.

Before we write the interpreter we also need a to define a stream for our input values. However, we can't use a data structure that stores values of a single type. There can be values of different types, furthermore we have to account for branching and recursion. To that purpose a session typed channel sounds like a good candidate. However, a session typed channel maintains two independent session types whereas we only need to keep track of one. This doesn't exclude `STChanT`, but it is probably a good idea to define a new data type that does exactly fit this specification. For that purpose we define a indexed state monad transformer.

```
newtype IxStateT m s r a = IxStateT { runIxStateT :: s -> m (a, r) }
```

Similar to STChanT, IxStateT helps us maintain and update the data structure that it carries. We define a GADT that is session typed as our data structure.

```
data STStream :: Cap -> Type where
  S_Send :: a -> STStream (Cap ctx r) -> STStream (Cap ctx (a :!> r))
  S_Recv :: a -> STStream (Cap ctx r) -> STStream (Cap ctx (a :?> r))
  S_Sel1 :: STStream (Cap ctx s) -> STStream (Cap ctx (s :+: r))
  S_Sel2 :: STStream (Cap ctx r) -> STStream (Cap ctx (s :+: r))
  S_Rec :: STStream (Cap (s ': ctx) s) -> STStream (Cap ctx (R s))
  S_Weaken :: STStream (Cap ctx s) -> STStream (Cap (t ': ctx) (Wk s))
  S_Var :: STStream (Cap (s ': ctx) s) -> STStream (Cap (s ': ctx) V)
  S_Eps :: STStream (Cap '[] Eps)
```

As is always the case when we define some session typed monad, we also define a set of functions for the interface.

```
s_Send :: Monad m => IxStateT m (Stream (Cap ctx (a :!> r))) (Stream (Cap ctx r)) ()
s_Send = IxStateT $ \(S_Send r) -> return ((), r)
s_Recv :: Monad m => IxStateT m (Stream (Cap ctx (a :?> r))) (Stream (Cap ctx r)) a
s_Recv = IxStateT $ \(S_Recv a r) -> return (a, r)
..
```

Finally we can define an interpreter that describes the full behavior of a session typed program.

```
run' :: (HasConstraint Show s, Monad m) =>
        STTerm m s r a ->
        IxStateT Identity (STStream s) (STStream r) (Output a)
run' (Send a r) = s_Send >> fmap O_Send $ run' r
run' (Recv c) = s_Recv >>= fmap O_Recv . run' . c
run' (Sel1 s) = s_Sel1 >> fmap O_Sel1 $ run' s
run' (Sel2 r) = s_Sel2 >> fmap O_Sel2 $ run' r
run' (Offer s r) = do
  str <- s_get
  s_Off1
  o1 <- run' s
  put str
  s_Off2
  o2 <- run' r
  return $ O_Offer o1 o2
run' (Rec s) = s_Rec >> fmap O_Rec $ run' s
run' (Var s) = s_Var >> fmap O_Var $ run' s
run' (Weaken s) = s_Weaken >> fmap O_Weaken $ run' s
run' (Ret a) = return $ O_Ret a
```

The definitions for all constructors mostly follow the same pattern; first we progress the session type using a session typed function, then we make a recursive call and finally

wrap the result into a constructor of `Output`. For `Offer` we have to do an extra bit of work. We don't have a second process that sends us a boolean to decide which branch to evaluate. We therefore need to evaluate both branches.

To define interpreters for the other two descriptions of the behavior of a session typed program, it mostly comes down to copying the definition of `run'` and removing generating the constructors that we do not want as output. To do this for an `Offer` is more difficult, because it involves two branches both with separate results. A trick here is to use a type family to insert a boolean into the session type of the stream, such that the input must contain an additional boolean that we use to select a branch to evaluate.

Using the newly defined interpreter we can almost purely evaluate `boolToInt`. The session type of `boolToInt` still expects some sort of continuation. If we only want to test a specific part of the session type program independently from the rest of the program, then we should be allowed to treat that specific part as if it were the entire session. We can therefore prepend the program with some initializing function that sets the context to empty and append an `end` that empties the session type.

```
initialize :: Monad m => STTerm m (Cap '[] s) (Cap '[] s)
run :: (HasConstraintST Show s, Monad m) =>
  STTerm m (Cap '[] s) (Cap ctx Eps) a ->
  Stream (Cap '[] (FilterSendST s)) -> Output a
run st str = fst $ runIdentity $
                runIxStateT (eval $ initialize >> st >>= end) str
```

Now we can use `run` to purely evaluate `boolToInt`.

```
> run boolToInt (S_Recv True $ S_Send S_Eps)
O_Recv True (O_Send 0 (O_Ret ()))

> run boolToInt (S_Recv False $ S_Send S_Eps)
O_Recv False (O_Send 1 (O_Ret ()))
```

5.2 Interactivity

The pure semantics interpreter is a useful tool for debugging and testing session typed programs, but it requires the user to prepare input for all possible paths that the program might follow. The combination of recursion with branching can make this task not so easy. Furthermore, it can be difficult to parse the output of the interpreter if it evaluates a fairly large program. An easier means to debugging is to evaluate a session typed program interactively. We can let the user supply the values for receives and let it select a branch to follow. Furthermore, we can let the user evaluate the program one step at a time, where one step equates to one session typed action,or have it abort evaluation early.

The interpreter for interactive evaluation takes a `STTerm` and a `Bool` as arguments, and returns a `Maybe` monad transformer.

```
interactiveStep' :: (MonadIO m, HasConstraints [Show, Typeable] s) =>
                    STTerm m s r a -> Bool -> MaybeT m ()
```

Since we want to be able to quit the interpretation at any point it should be possible to ignore all computations after the point where we stopped. The semantics of this is similar to how the `Maybe` monad works. It continuously applies functions to the value in a `Just` until the result of such an application is a `Nothing`. Then any function applications after that are ignored.
First we'll give a definition for the `Send` pattern.

```
interactiveStep' s@(Send a r) b = do
  print s b
  waitStep
  interactiveStep' r b

print (Send a _) True = liftIO $ "> Send value " ++ show a
print _ False = return ()
```

For `Send` there isn't anything interesting to do. So all we do is print a message to console that we have now evaluated the `Send` together with the value that is being send, but only if the user wants something to be printed, which is determined by the boolean argument. It should now also be clear why we need `MonadIO` and `Show` constraints. The function `waitStep` may either continue evaluation or will stop it depending on the user's input. It prints the following message:

```
"?> Press n to continue or q to quit."
```

In the case that n is pressed it simply returns a `Just ()` and otherwise we return `Nothing`.

The definitions for the other constructors of `STTerm` are similarly defined, except for `Recv` and `Offer` where we require input from the user.

```
interactiveStep' st@(Recv r) b = do
  print st b
  ma <- liftIO $ fmap readMaybe getLine
  case ma of
    Nothing -> interactiveStep' st b
    Just a -> waitStep >> interactiveStep' (r a) b
interactiveStep' st@(Offer s r) b = do
  print st b
  br <- liftIO getLine
  case br of
    "Left" -> waitStep >> interactiveStep' s b
    "Right" -> waitStep >> interactiveStep' r b
```

```
    _ -> interactiveStep' st b

print r@(Recv _) _ = liftIO $ putStr $
    "?> Enter a value of type " ++ typeShow r ++ ": "
  where typeShow :: forall m ctx a r k b. Typeable a =>
                    STTerm m ('Cap ctx (a :?> r)) k b -> String
        typeShow _ = show $ typeRep (Proxy :: Proxy a)
print (Offer _ _ ) _ = liftIO $ putStr $ "?> Enter Left or Right: "
```

For `Recv` we ask the user to produce a value of the right type. To help the user understand what kind of value we are expecting we also print out the name of this type. Doing so requires the `Typeable` constraint that we also added to the constraint set in the type of `interactiveStep'`. If the user ends up giving us an incompatible type, then we simply ask for another value. Otherwise we continue.
Also for `Offer` do we require user input. Normally, we would receive a boolean from the other process in the session that decides the branch to evaluate. Since that process for this interpreter does not exist, we need the user to manually do it for us.

We provide a small wrapper function that calls `interactiveStep'` and unpacks the result from the maybe monad transformer.

```
interactiveStep :: (MonadIO m, HasConstraints [Show, Typeable] s) =>
                   STTerm m s r a -> Bool -> m ()
```

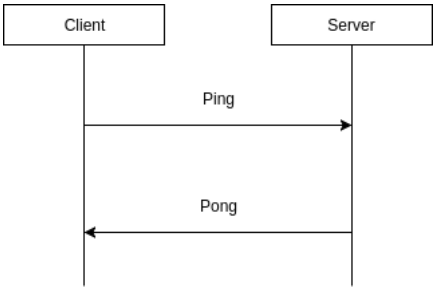
Then to conclude this section we give an example interaction. The following shows a session typed program (left) and a possible interaction (right).

```
prog = do
  x <- recv
  send x
  offer (end ()) $ do
        y <- recv
        end y
```

```
?> Enter a value of type Int: 5
?> Press n to continue or q to quit: n
Send value 5
?> Press n to continue or q to quit: n
?> Enter Left or Right: Right
?> Press n to continue or q to quit: n
?> Enter a value of type String: "Test"
?> Press n to continue or q to quit: n
Result: "Test"
```

5.3 Visualization

A great method for developing a communication protocol is designing a UML that describes the protocol as a sequence diagram. The sequence diagram gives a visualization of the protocol, which is often easier to understand then if the protocol were written out in words or in code. For example, a sequence diagram of the ping-pong protocol can be seen below. In this diagram there are two actors, the client and the server. The vertical lines below the actors represent time. The outgoing and incoming arrows originating from the vertical lines represent an action at that specific moment in time. We do not



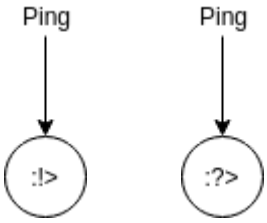
need to pin-point an exact moment for these actions as we only care about the order of the actions as they occur. The first action that occurs is the client sending a 'ping request' to the server. Upon receiving this message the server continues by sending a 'pong response' back to the client, after which both the client and the server have finished the protocol. A simple protocol like this one does not really need a visualization to make it any easier to understand, but it can prove to be very beneficial for more complex protocols.

We want to write an interpreter that visualizes the session type of a session typed program as a sequence diagram. Technically, this isn't an interpreter for session typed programs since we can always only pass the session type to it using a `Proxy`. Nonetheless, we want to define a function that takes a `STTerm` and creates a visualization of the session type.

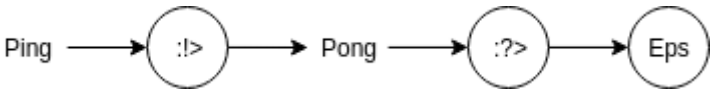
```
visualize :: forall m s r a. STTerm m s r a -> IO ()
visualize _ = visualizeP (Proxy :: Proxy s)
visualizeP :: Proxy s -> IO ()
```

To visualize session types we could potentially use a sequence diagram such as the one we used for the ping-pong protocol. Then we would replace 'ping' and 'pong' with the name of the types of values being sent and received. A session type protocol always consists of only two actors, so we replace 'client' with 'Process A' and 'server' with 'Process B', denoting the processes in the sessions. We also need to consider how we embed branching and recursion. Specifically branching makes this type of sequence diagram difficult to use. Consider this relatively simple session type that is an expanded version of the ping-pong protocol: `Ping :!> (Pong :?> Int :!> Eps) :&: (Ping :?> Bool :!> Eps)`. The server may now send back either a ping or a pong. We could denote this by a single arrow from the server to the client that is annotated with `Pong / Ping`. Then afterwards the client needs to send an `Int` or a `Bool` respectively for the type of message received. Again we could draw an arrow to the server that is annotated by both, but then the diagram would indicate that it is possible for the client to receive a `Pong` and then send a `Bool`, which according to the protocol should not be possible. It is apparent that having only two lines for potentially many more branches does not work. We could split the line whenever we branch, but that would mean arrows often have to cross lines without interacting with it, which hurts readability of the diagram.

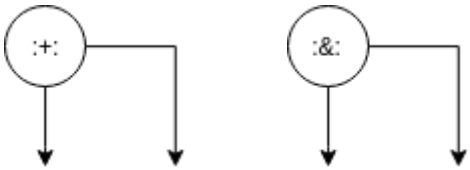
Instead of a sequence diagram we can opt for a more general and simpler visualization of a session type. This type of visualization consists of nodes that are (partial) session types and arrows between the nodes that connect these session types. Let's first consider sending and receiving. The diagrams displayed below, give a visualization of the



following two session types `Ping :!>` and `Pong :?>`. Now an arrow no longer represents a send or receive, instead it 'connects' parts of a session type. In this case it connects the type of message to a send or receive session type. For each session type we can make building blocks that represent the session type and then connect these blocks to form a visualization of the entire session type. For example, we can connect these two diagrams to form a visualization of the ping-pong protocol. We also added another node

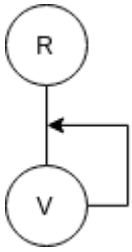


for the session type `Eps`. This time the diagram is displayed horizontally as it does not matter whether we use a horizontal or vertical orientation. However, we should try to be consistent in which orientation the diagram is displayed such that it does not generate unnecessary confusion as to whether a turn in an arrow means anything.



The diagrams for branching start with a node for the session type. It has two outgoing arrows. The first arrow that goes down connects the first branch to the session type and the second arrow, connecting the second branch, goes horizontally to the right, where it then makes a turn downwards. Here a turn in an arrow actually does mean something; an arrow connecting to the second branch of a branching session type. The exactly place where the second arrow turns is determined such that it does not cross with any node or arrow belonging to the first branch.
The last session types we need to consider are the recursive session types. All of these can be represented by a single node.

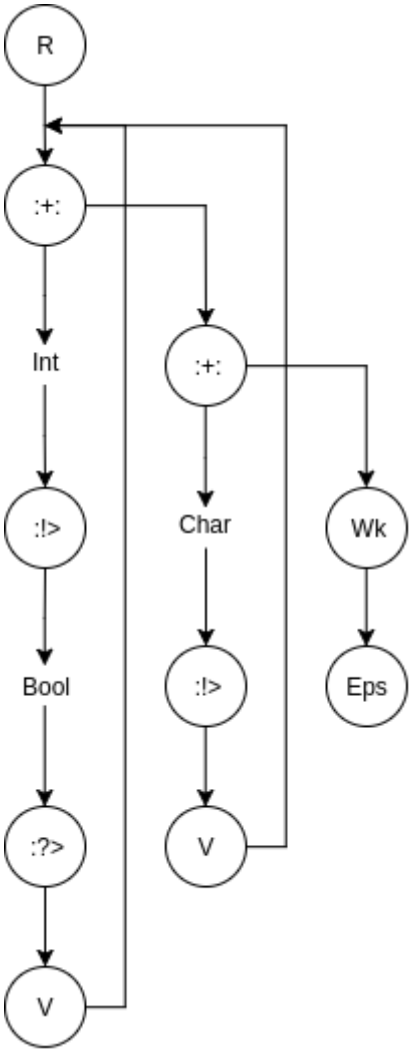
Recursion and weaken connect to a session type as you would expect. However, a recursion variable should not connect to a session type that is located below him, after all it recurses back to just after an `R`. So for `V` we should trace an arrow to a node that is right below the `R` that corresponds to this `V`. What we actually do is connect it to the arrow that is in between `R` and the session type below it.



If the `V` were to connect to a node that is a branching session type, then it would already have a right outgoing arrow, which could overlap with the arrow coming from `V`. To avoid this overlap, we actually make the node with `V` trace an arrow to right between the corresponding `R` node and the node below it. It is still possible for an arrow coming from a `V` node to intersect with the branching arrow. We believe some intersection of arrows is not necessarily bad for the clarity of the diagram.

We finish this section with an example diagram of the following session typed

```
R ( Int :!> Bool :?> V
:+: Char :!> V
:+: Wk Eps
)
```



5.4 Normalizing session types

A function can often be written in more than one way. For example, a function that adds two integers can be written in two ways: `f x y = x + y` and `f' x y = y + x`. It should not matter whether we use `f` or `f'` to add two integers. In fact, we should be able to replace `f` with `f'`, wherever `f` is used. Similarly, we should also be able to replace a session typed program with another as long as they are isomorphic. For example

```
prog1 :: STTerm m (Cap ctx ((Eps :&: Eps) :&: Eps) (Cap ctx Eps) ()
prog1 = offer (offer (end ()) (end ())) (end ())
prog2 :: STTerm m (Cap ctx (Eps :&: (Eps :&: Eps)) (Cap ctx Eps) ()
prog2 = offer (end ()) (offer (end ()) (end ()))
```

For `prog1` and `prog2` to be isomorphic, we should be able to rewrite `prog1` into `prog2`

and prog2 into prog1. Since all branches in both programs only lead to ending the session, this should be possible without changing the meaning of either program.

Often you can rewrite one session typed program into another, but you cannot do the reverse. For instance, prog3 can remove its send constructor, but prog4 cannot add a send constructor without knowing what the value of the first argument must be.

```
prog3 :: STTerm m (Cap ctx (Int :!> Eps)) (Cap ctx Eps) ()
prog3 = send 5 >>= end
prog4 :: STTerm m (Cap ctx Eps) (Cap ctx Eps) ()
prog4 = end ()
```

Session types only tells us which constructors to place where, but other values are only represented by their types in the session type. So we should never only consider session types to determine whether a session typed program is isomorphic to another.

We were able to find two types of rewriting that we can apply to session typed program and that preserves the isomorphism: associative rewriting and elimination of unnecessary recursion. Before we explain these two types of rewriting it is important to understand why rewriting session typed programs is important or even necessary. Consider a possible dual program to prog2 named dprog2. We want to run dprog2 together with prog1 in a session. However, we cannot do this because their session types are not dual. We know this because the session types of prog1 and prog2 are not equal. Though we know that prog1 can be rewritten to prog2, which would allow us to spawn a session consisting of two programs with non-dual session types. Since, prog2 can also be rewritten into prog1 we have to determine a normal form, such that a session typed program will be rewritten until it is in normal form. For example, if prog2 is determined to be in normal form then no rewriting will be done. However, prog1 will be rewritten until it is in normal form and thus equal to prog1.

5.4.1 Associative rewrite rules

The rewrite rule that we employed to rewrite prog1 into prog2 was the associative rewrite rule. Basically we take a session type that contains a nesting of branches and rewrite it in such a way that it becomes right associative.

```
(s :+: r) :+: t >>> s :+: r :+: t
(s :&: r) :&: t >>> s :&: r :&: t
```

Note that both :+: and :&: are right associative by definition. We can therefore leave out the brackets in their normal form. We use a type family to apply this transformation. We also have to rewrite the AST. A simplified function for rewriting looks as follows:

```
rewrite (Send a r) = Send a $ rewrite r
rewrite (Recv r) = Recv $ \a -> Recv (rewrite $ r a)
rewrite (Sel1 (Sel1 s)) = Sel1 s
```

```
rewrite (Sel1 ((Sel2 r)) = Sel2 (Sel1 r)
rewrite (Sel2 t) = Sel2 (Sel2 t)
rewrite (Offer (Offer s r) t) = Offer s (Offer r t)
rewrite (Rec s) = Rec $ rewrite s
rewrite (Weaken s) = Rec $ rewrite s
rewrite (Var s) = Var $ rewrite s
rewrite (Ret a) = Ret a
rewrite (Lift m) = Lift $ m >>= return . rewrite
```

For `Offer` we only need one pattern as we can immediately rewrite all three branches. With selection you can only choose a single branch, so we need three patterns. For all other patterns we simply need to traverse the AST. A single rewrite to an AST might not be enough to make it normal, therefore we also define a function that repeatedly applies `rewrite` until a fix point has been reached.

5.4.2 Eliminating recursion

Another transformation we can apply is removing unneeded recursions and weakens. If we have introduced recusion, but there is no recursion variable then there is also no reason to introduce the recursion in the first place. We would like to prove that a session type program that introduces recursion, but does not recurse is equal to the same session type program that does not not introduce the recursion.
It is not very straightforward to decide whether we can remove a `R` or a `Wk`. We know a `R` should not be removed if it matches with a `V`, but that depends on how many `R's` and `Wk's` can be found on the path to that `V`. A `Wk` should be removed if it does not help a `R` reach some `V`. To make it easier to understand we will consider several session types and explain why we transform them the way we do. All of these examples are completely trivial; they don't do any sending or receiving, but the existence of these other session types is irrelevant for deciding how we should transform these session types.

(1) `R V >>> R V`

In the first case we have simple recursion; we introduce recursion and recurse. Although no session typed actions occur after introducing the recursion and before the recursion variable, we cannot assume that no unsession typed computations are done. Therefore we do not do anything.

(2) `R Eps >>> Eps`

In this case, we introduce the recursion but never make use of it. As such, we do not need to introduce recursion.

(3) `Wk Eps >>> X`

This session type is actually impossible to construct using only the combinators provided by the library. If we were to normalize it, we would normalize it to be `Eps`. However, to rewrite it back to `Wk Eps` one must know about the session type in the context. Without

introducing recursion there is no way for us to know of that session type. We therefore do not accept this program to be normalized.

(4) `R (Wk Eps) >>> Eps`

Now we do introduce the recursion before weakening the scope, but we do not use a recursion variable. We can therefore rewrite the session type. If we were to reverse the rewriting then we would know the session type in the context must be `Wk Eps`.

(5) `R (R (Wk V)) >>> R V`

Here we have a similar situation. Now the inner `R` does not match a `V` so we remove it. Consequently, we also remove the `Wk`. If we hadn't done so, then the outer `R` would no longer be matching any `V`.

(6) `R (R (R (Wk V :+: Wk (Wk V)))) >>> R (R (V :+: Wk V))`

The last case is somewhat more complex. By adding branching we can make multiple `R's` match a `V`. We quickly see that the most inner `R` does not match anything, so we get `R (R (Wk V :+: Wk (Wk V)))`. Now it looks like the `Wk` in the first branch helps the outer `R` reach the `V` in the first branch. We must not forget that we had removed an inner `R`. So we should still remove a `Wk` from both branches. It is important that if we remove an `R` we also determine any `Wk` that should be removed, such that the outer `R` keep matching the same `V` as they did before the transformation. We therefore remove a single `Wk` in each branch.

Similar to `rewrite` we define a type class that exposes a function that eliminates unnecessary recursion. We also won't give a definition for it, because it ended up being quite complex and it would be very difficult to explain how it works. Although, on a higher level it works as described in the previous paragraph. Ultimately, the combination of `rewrite` and a function named `elimRec` gives rise to our normalizer.

```
normalize :: (Rewrite s s', ElimRec s' s'') => STTerm m s r a -> STTerm m s'' r a
```

5.4.3 Distributive and factorize

This section doesn't describe a new transformation rule, but rather one (or actually two) that we initially had expected to hold but ultimately didn't. We thought that we could apply the distributivity rule to session types as well. We take a session type that occurs right before a branching and then move it to the front of both branches.

```
a :!> (s :+: r) >>> a :!> s :+: a :!> r
a :?> (s :+: r) >>> a :?> s :+: a :?> s
```

At type level this seemed to make a lot of sense, but problems occurred when actually trying to make this rewriting happen.

```
rewrite (Send a (Sel1 s)) = Sel1 (Send a s)
rewrite (Send a (Sel2 r)) = Sel2 (Send a r)
-- is incorrect
rewrite (Recv c) = Recv $ \a -> case c a where
  (Sel1 s) -> Sel1 $ rewrite s
  (Sel2 r) -> Sel2 $ rewrite r
```

When branching is preceded by a send, it becomes possible to distribute the send to both branches. However, doing the same for a receive is impossible. The `Recv` constructor carries a continuation that given an argument will return the rest of the AST that includes the branching constructors. This argument can only be produced by defining a new `Recv` constructor. However, if we first define the `Recv` constructor, then we are not rewriting the way we intended to; the branching constructors will still come after the `Recv`. At first we thought this was merely a limitation of using a continuation for `Recv`, but this appeared not to be the case. Consider the next program:

```
prog = do
  n <- recv
  go n
where
  go 0 = sel1 >> end "B1"
  go n = sel2 >> end "B2"
```

Which branch we choose depends on the value returned by `recv`. It would be impossible to do this if we were to make a transformation that places `recv` after `sel1` and `sel2`. Apparently we can't make this transformation, not because of a limitation of continuations, but because it simply is impossible to do this.

The problem described above clearly rules out a possibility for using distributivity as a rewrite rule, but then perhaps we can factorize instead. The answer to that question is a clear no. Factorizing appears to exhibit the dual of the problem that we had with distributivity.

```
a :!> s :&: a :!> r >>> a :!> (s :&: r)
a :?> s :&: a :?> r >>> a :?> (s :&: r)
```

Factorizing does the opposite of distribution. We take a session type from a selection or offering that is common in both branches and have it precede the branching.

```
rewrite (Offer (Send a s) (Send b r)) = Send ? (Offer s r)
rewrite (Offer (Recv s) (Recv r)) = Recvr $ \a -> Offer (Recv (s a)) (Recv (r a))
```

To make this rewrite rule work we have to merge two sends, but each `Send` contains a value that will later be used. We know that the types of these values are equal, but the values themselves are not necessarily so. That makes it impossible for us to pick one.

For both factorization and distribution the problems lies in that we want to generate

a structure that is statically known, but contains values that depend on other dynamically known values. Either these dynamically known values do not exist yet at the time of rewriting or we cannot decide which to use.

In this chapter we have introduced exactly four interpreters other than the interpreter for the Cloud Haskell semantics. Specifically, we defined an interpreter for purely debugging a session typed program, impurely stepping through a session typed program, visualizing session types and normalizing session typed programs. In the next chapter we will look at some optimizations that we can implement to hopefully somewhat counter the potential performance hit that using session types costs us.

6 Optimizations

6.1 Nested branches

The current branching implementation is very inefficient. Whenever we make a choice to go into a specific branch the other process in the session has to be notified. This is done by sending a boolean (or using our custom data type from chapter 4). If we take the left branch, then we send `True` and if we take the right branch we send `False`. This notification procedure is necessary to keep the session synchronized. If we do not send a boolean or some other data type to indicate which branch we take, then the other process has no way of knowing which branch to evaluate. However, the current inefficiency of this procedure is in how it works with nested branches. Consider this (partial) session type:

```
s :+: r :+: t :+: k :+: i
```

For this example it doesn't matter what each branch does. It is only necessary to understand how we choose a specific branch. If we choose `s` then we send `True`, but if we want to choose `r` then we first have to send `False` to specify that we do not want the first branch followed by `True`. For each right associated nested select we have to send an additional boolean and this is quite inefficient. Sending messages over the network is slow compared to local computation and these messages are only used for synchronization and not actual information relevant to the program itself.

An optimization we could implement is to send a single integer instead of a sequence of booleans. The value of the integer denotes the branch we're choosing, where `0` refers to the first branch. In the example above to choose `s` we would send `0` and to choose `i` we would send `4`. We implement this optimization in the `eval` function, where in the select patterns we have to count the number of right nested selects. Unfortunately, when traversing a nesting of right associatively nested selections it is difficult to determine which branch is the very last branch, especially when the traversing is guided by a type family. To show what I mean we will try to write a function that unfolds a selection.

```
unfoldSelect :: .. => Int -> STTerm m s r a ->
                      STChanT m (SendInsInt s :*: RecvInsInt s)
                                (SendInsInt r :*: RecvInsInt r) a
unfoldSelect k (Sel1 s) = sel1ChanM >> sendChanM k >> eval s
unfoldSelect k (Sel2 r) = sel2ChanM >> unfoldSelect (k + 1)
type family SendInsIntST s where
  ..
  SendInsIntST (s :+: r) = (Int :!> SendInsIntST s) :+: SendInsIntST r
```

We use the type family `SendInsIntST` to force sending an integer in the `Sel1` pattern. The integer that is sent should be the index of the chosen branch. To determine what this index is, the function `unfoldSelect` counts the number of times that a `Sel2` is nested in another `Sel2`. With the idea that upon reaching a `Sel1` we have selected a branch with known index. We do not send an integer in the second branch of a select, otherwise we would have to send an integer each time we unfold a `Sel2`. The function assumes that a nesting of selections will always end with a `Sel1`, but this is actually never the case. `Sel1` is indeed used to select a branch, but you cannot use it to select the very last branch. This is done using `Sel2`. So now we have the problem that we both do and do not want to send an integer in the `Sel2` pattern. It is possible to add extra patterns by pattern matching on the argument to `Sel2`. If that argument is another `Sel2`, then we know we have not reached the last branch and otherwise we know it is. However, doing this amount of extra work seems unnecessary and perhaps the current implementation of selection and offering is limited in a way. We would prefer to have a structure where we can easily identify the last branch or one that has a separate constructor for selecting a branch and traversing over a nesting of branches.

A suitable structure seems to be a heterogeneous list. It is typed, such that it can be combined with our session types and it has a separate constructor for the empty list. To embed a heterogeneous list in session types we redefine `:+:` and `:&:` in terms of `Sel` and `Off` respectively.

```
data ST = .. | Sel [ST] | Off [ST] | ..
```

Both `Sel` and `Off` carry a type list of session types, where `Sel` is used for selection and `Off` for offering. We also add constructors to `STTerm` and remove the now unneeded constructors.

```
data STTerm :: .. where
  ..
  SelN1 :: STTerm m ('Cap ctx s) r a -> STTerm m ('Cap ctx (Sel (s ': xs))) r a
  SelN2 :: STTerm m ('Cap ctx (Sel (k ': xs))) r a ->
           STTerm m ('Cap ctx (Sel (s ': k ': xs))) r a
  OfferZ :: STTerm m ('Cap ctx s) r a -> STTerm m ('Cap ctx (Off '[s])) r a
  OfferS :: STTerm m ('Cap ctx s) r a ->
            STTerm m ('Cap ctx (Off xs)) r a ->
            STTerm m ('Cap ctx (Off (s ': xs))) r a
```

Now any nested selection will end with a `SelN1` that is typed with a non-empty list. It is non-empty, because the selected branch is not always the last branch. The constructor `SelN2` is still used to traverse over the selection and that is also all it can be used for. For that reason it takes a `STTerm` that is typed with another selection, such that it must take another `SelN2` or `SelN1`. For the same reason its argument must be typed with a non-empty list.
Unlike selection, we have to define all branches for an offering. We could make its structure identical to that of the heterogeneous list, However, we would not be able to give a

bind definition for `OfferZ`, that is if it were to represent the empty list and did not have any arguments.

To completely replace `:+:` and `:&:` with `Sel` and `Off` respectively, we have to modify most of the existing code. We're not going to show all changes made, but we will show the changes that are relevant to the optimization that we're trying to implement.

First we will modify the `Dual` type family. As we will keep doing, we remove the patterns for selection and offering and add two for the new session types.

```
type family Dual s = r  r -> s where
  ..
  Dual (Sel xs) = Off (MapDual xs)
  Dual (Off xs) = Sel (MapDual xs)
  ..

type family MapDual s = r  r -> s where
  MapDual '[] = '[]
  MapDual (s ': xs) = Dual s ': MapDual
```

We can compute the dual for both session types by replacing the outer constructor by its dual and then applying `Dual` to the contents of the list contained in the session types. Next we change the combinators for selection and offering that work on session typed channels, such that in `eval` we can still choose to go into a specific branch. Though, these are almost no different from their old definition. What is more interesting are the type families used to guide the types for `eval`.

```
  SendInsIntST (Sel xs) = Sel (MapSend (MapInsInt xs))
  SendInsIntST (Off xs) = Off (MapSend xs)

  RecvInsIntST (Sel xs) = Sel (MapRecv xs)
  RecvInsIntST (Off xs) = Int :?> Off (MapRecv xs)

  RecvInsIntST' (Off xs) = Off (MapRecv )

type family MapSend xs where
  MapSend '[] = '[]
  MapSend (s ': xs) = SendInsIntST s ': MapSend xs
type family MapInsInt xs where
  MapInsInt '[] = '[]
  MapInsInt (s ': xs) = (Int :!> s) ': MapInsInt xs
type family MapRecv xs where
  MapRecv '[] = '[]
  MapRecv (s ': xs) = RecvInsIntST ': MapRecv xs
```

For the send port we need to send an integer after selecting a branch. We use the type family `SendInsIntST` to do this for us. In the selection pattern it calls `MapSend`, which prepends each branch with `Int :!>`. That does not mean after every `SelN2` we must send an integer. This is because `SelN2` stays within the `Sel` session type and does not select a branch. Only after `SelN1` do we extract a session type from the selection that is now prepended with the session type for sending an integer. We can therefore traverse and count as many `SelN2` as we want, before having to send the final count. This is implemented in the function below.

```
unfoldSelect (SelN1 s) k = sel1ChanM >> sendChanM k >> eval s
unfoldSelect (SelN2 r) k = sel2ChanM >> unfoldSelect r (k + 1)
```

For an offer it works slightly different. Now we need an integer that we can use to index a branch. Since we need this integer before doing any traversing, we preprend the offer session type with `Int :?>`. In the `unfoldOffer` we will assume that we have already received this integer. It is therefore important that in the type signature of `unfoldOffer` we do not modify the session type in any way, which is different from selection. In `unfoldOffer` we simply decrement the counter while traversing the nesting of branches until it becomes `0`. Note that reaching `OfferZ` would be incorrect, since it does not contain An offering on the other hand must receive an integer before traversing the offering. In the type family this is easily done by prepending a `Int :?>` to an `Off`. Then in our unfolding function we keep decrementing the received integer until it becomes `0` at which point we evaluate the branch that we had unfolded to.

```
unfoldOffer (OfferZ s) 0 = off1ChanM >> eval s
unfoldOffer (OfferN s xs) 0 = off1ChanM >> eval s
unfoldOffer (OfferN s xs) k = off2ChanM >> unfoldOffer xs (k - 1)
```

The receiving of the integer is done in `eval` itself. We have also adjusted the type of `unfoldOffer` just slightly compared to that of `eval` such that we do not need to receive an integer for every offering constructor.

```
eval s@(SelN1 _) = unfoldSelect s 0
eval s@(SelN2 _) = unfoldSelect s 0
eval o@(OfferZ _) = recvChanM >>= unfoldOffer o
eval o@(OfferN _ _) = recvChanM >>= unfoldOffer o
```

6.2 Codensity

Another potential source of inefficiency is the structure of a free monad. A free monad may exhibit quadratic complexity by repeated use of binds that results in a left associative nesting of the free monad. This is no different for `STTerm` that is an indexed free monad. In this section we will try to reduce the quadratic complexity to a linear one by using the Codensity monad [22].

6.2.1 Quadratic complexity vs Linear complexity

Let's start by illustrating why a free monad may exhibit quadratic complexity. Consider a simplified instance of `STTerm` that consists of only `Send` and `Ret` and is also not session typed.

```
data STTerm a where
  Send :: a -> STTerm a -> STTerm a
  Ret :: a -> STTerm a
```

We could write a simple program that sends a character `n` times in two different ways.

```
sendMany1 0 = return ()
sendMany1 n = send 'c' >> sendMany1 (n - 1)

sendMany2 0 = return ()
sendMany2 n = sendMany2 (n - 1) >> send 'c'
```

Surprisingly, if you were to benchmark both versions then the first should always be the fastest of the two. This is because the recursive call in `sendMany1` occurs under a right nested bind. On the other hand, the recursive call in `sendMany2` is followed by a bind resulting in a left nested bind. We can verify this if we supply both functions with an argument and expand the recursive call.

```
sendMany1 3 = send 'c' >> (send 'c' >> (send 'c' >> (return ())))
sendMany2 3 = (((return ()) >> send 'c') >> send 'c') >> send 'c'
```

The left associative nesting in `sendMany2` causes quadratic complexity, because for each repeated call of `>>` the up to that point generated AST needs to be traversed. We can see this in the definition of bind.

```
Send a r >>= f = Send a (r >>= f)
Ret a >>= f = f a
```

If we have a nesting of `n Sends` that we bind with a function, then the entire AST would need to be traversed until a `Ret`, which is replaced with the result of applying the argument of `Ret` to the function, thereby appending the from the function generated AST to the nesting of `Sends`. For one such bind we have linear complexity. If we do a bind `n` times we get quadratic complexity.
In `sendMany2` the first recursive call that we can fully evaluate is `sendMany2 0`, which returns a `Ret ()`. Every evaluation of a recursive call after this replaces the remaining `Ret ()` in the so far generated AST with `Send 'c' (Ret ())`. As a result each recursive call ends up appending a `Send 'c' (Ret ())`, but since the `Ret ()` is always placed at the end of the AST, the entire AST will first have to be traversed.
The right associative nesting in `sendMany1` ensures that the left argument of bind is at most `Send 'c' (Ret ())`, such that a bind takes a constant amount of time. Then the repeated use of bind results in linear complexity.

6.2.2 Implementing codensity

Voigtländer describes [22] a method to turn any free monad representation into an alternate one that does not require traversing an entire AST for each bind. He defines the Codensity monad alongside two functions that connect an ordinary free monad and its alternate representation.

```
data C m a = C { runC :: forall b. (a -> m b) -> m b }

rep :: Monad m => m a -> C m a
rep m = C $ \h -> m >>= h
abs :: Monad m => C m a -> m a
abs (C h) = h return
```

The function `rep` takes a monad and turns it into the alternate representation, `abs` does the reverse and the composition of the two should always equal the identity function. Essentially you can use `rep` to create an API in the alternate representation of which we know it to have reduced complexity. Then when we wish to return to our normal representation we can simply apply `abs` without returning to a quadratic complexity.

The reason the codensity monad does not require traversing the entire AST can be found in the definition of bind.

```
data Monad m => Monad (C m) where
  return a = C $ h a
  (C h) >>= f = C $ \c -> h $ \a -> runC (f a) c
```

The codensity monad always has a continuation that points to the bottom of the AST. This continuation can be used to efficiently append two ASTs by passing the second AST to the continuation, thereby constructing a new AST in constant time, and constructing a new continuation that points to the bottom of the second AST and therefore also of the entire AST. This procedure is similar to trying to append two linked lists. If we do not have direct access to the pointer that points to the bottom of the first list, then we need to traverse it causing linear complexity for each append. Having direct access to this pointer means you can do it in constant time. Furthermore after appending, the bottom pointer of the second list should be directly accessible allowing efficient appending for the newly constructed linked list.

6.2.3 Turning STTerm into Codensity

As is usually the case we can't use the codensity monad defined in the previous section for `STTerm`, because it is not session typed. We create a new indexed codensity monad by adding two more parameters that will be used for indexing.

```
newtype IxC m s r a = IxC {
  runIxC :: forall b. (a -> m (Fst r) (Snd r) b) -> m (Fst s) (Snd s) b
  }
```

The type of `runIxC` is actually very similar to that of `Recv`. The difference is that for `Recv` we are working with a specific session type, which also tells us what its continuation session type is. The continuation session type of `a :?> r` is `r`. This `r` is also the initial state of the `STTerm` argument of `Recv`. For `runIxC` it is not possible to determine what the initial state of its continuation should be, because the parameterized session types are polymorphic. With no way to determine what the initial state of the continuation should be, we have to parameterize it. That is the reason why we once more make use of `Prod` to describe more than two session types. It should be noted that we actually only need to describe three different session types. The Similar to `Recv`, the final state of the continuation determines the final state of the resulting monad.

We will now give a couple function definitions for the different session types. We do not show the indexed monad instance for `IxC`, since it is no different from the non-indexed version.

```
type STTermC m s r a = IxC (STTerm m) s r a

send :: a -> STTermC m (Cap ctx (a :!> r) :*: r') (Cap ctx r :*: r') ()
send a = IxC $ \h -> Send a $ h ()
```

The definition of `send` is quite similar to that of the `send` function for a `STTerm`. Insead of a `Ret ()` we pass `h ()`, the continuation, as an argument to the `Send` constructor. Its initial state is derived from the send session type and has a final state of `r'`. For the reason given above, this is then also the final state of the `Send` constructor.
Some of the session type primitives require arguments.

```
offer :: STTermC m (Cap ctx s :*: t) k a
      -> STTermC m (Cap ctx r :*: t) k a
      -> STTermC m (Cap ctx (s :&: r) :*: t) k a
offer (IxC f) (IxC g) = IxC $ \h -> Offer (f h) (g h)
```

In the case that we are given an argument we have to pattern match on the codensity monad. In doing so, we gain access to the underlying functions that can be given a continuation. The same continuation is passed to both branches, such that any function composition occuring after the offer is applied to both branches. Once more, the definition is similar to the `offer` over `STTerm`. Now we only have to think about how and where we apply the continuation.

Now that we should understand how these functions are typed we can also redefine `rep` and `abs`.

```
rep :: IxMonad m => m s r a -> IxC m (s :*: r) (r :*: r) a
rep m = IxC $ \h -> m >>= h
abs :: IxMonad m => IxC m (s :*: r) (r :*: r) a -> m s r a
abs (IxC f) = f $ \a -> return a
```

In `rep` we let the continuation traverse the AST, such that it will point to the bottom of the AST. For `abs` we need to define the continuation that is pointing to the bottom of the AST. We can make the continuation simply return its argument, such that the continuation also does not modify the state. As a result of defining the continuation, instead of passing it around like in `send` and `offer` do, all passed around continuations can now be evaluated and thereby generating the AST.

As a final touch to this section we will try to simplify the type signature of `send`. Since we know that the final state of the continuation and the final state of the resulting monad in the codensity monad must be the same, there should be no reason to have to write both. We can write a simply type synonym that extracts this duplication.

```
type STTermC m s k r a = IxC (STTerm m) (s :*: r) (k :*: r) a
```

So now `STTermC` is given three parameters of kind `Cap`. The first two describe the initial states of the resulting monad and the continuation respectively. The third parameter then describes the final state of the codensity.

In this chapter we have introduced two types of optimizations. One relates to the communication required for branching, where instead of a sequence of booleans a single integer can be send. For that purpose we had modified the branching session types to take a list of types. The second optimization is Codensity that can reduce quadratic complexity introduced by left associatively nested free monad ASTs to linear complexity. This was the last chapter that either introduced new features or simply improved the library. In the next and final chapter we will perform some benchmarking tests.

7 Benchmarking

We benchmarked the performance of session typed programs. Specifically we measured the gains or losses to be had from optimizing and normalizing session typed programs and we compared the performance of a Cloud Haskell program and its session typed equivalent.
We have two ways of evaluating a session typed program that gives us an immediate result without requiring any intervention. First of all we can evaluate a session typed program as a Cloud Haskell program (using `eval`). The other option we have is using the teletype interpreter for purely evaluating session typed programs. In our benchmarking we have preferred to use the teletype interpreter, because there is no overhead required for running a process and it is devoid of any delay incurred from local and remote transmissions of messages. This allows us to give a more accurate reading on the performance impact caused by optimizing or normalizing a session typed program. We cannot solely use the teletype interpreter for our benchmarking; Cloud Haskell programs cannot be purely evaluated. So for this type of benchmarking we will only consider session typed programs and any optimizations that may apply to it. For the other type of benchmarking we use `eval` to compare the performance of a Cloud Haskell program, its session typed equivalent and any optimizations that we may apply.

The benchmarks involving the teletype interpreter consisted of four different programs that were benched under four different settings each. With a setting we mean whether any optimizations were applied to the program. These settings are described below together with their abbreviations.

- ST : Session typed program with no optimizations
- ST-N : Session typed program that has been normalized
- ST-C : Session typed program using codensity
- ST-(N+C) : Session typed program that has been normalized and uses codensity

The benchmarks themselves were executed using the criterion library. It runs a specific benchmark 1000 times, returns the average time and also gives us an estimate of how reliable the results were (how many outliers were there). Since no IO was involved, all computed results were deemed accurate.

The first program that we benchmarked consisted of repeating a send of an integer followed by a receive of an integer exactly 10 times. The receives integers were summed and returned as the result of the program.

ST	ST-N	ST-C	ST-(N+C)
4.73 µs	5.92 µs	3.81 µs	4.53 µs

With a base time of 4.73 µs we see that normalizing this program worsens the performance. This can easily be explained: the given program is already in normal form. So any additional computation that we do only worsens the performance. The next result if from using codensity, which also as expected improves the performance. Reducing the quadratic complexity to linear complexity clearly affects the performance in a positive way. It is possible that if we were to increase the size of the benchmarked program that this difference in speed becomes larger. The final result is the combination of normalizing and codensity. While still better than the base time, the inclusion of normalizing clearly reduces the improvements made by codensity.

The next benchmarked program does the same as the previous one, but has increased the size of the number of branchings. There is now a right associative nesting of 10 branches, where in the very last branch the send and receive are done.

ST	ST-N	ST-C	ST-(N+C)
14.8 µs	19.3 µs	9.97 µs	13.6 µs

The results are obviously higher than the results of the previous benchmark, but they follow the same pattern. Normalizing worsens performance, while codensity improves it. More interestingly is if were now compare it to a similar program where the nesting of branches is left associative.

ST	ST-N	ST-C	ST-(N+C)
16.0 µs	9.81 µs	11.5 µs	5.29 µs

This time we see that normalizing drastically improves the performance. Also codensity performs as expected. However, this time the improvements made by normalizing and codensity only further improves the speed making this version almost 3x as fast as the base program.

The final purely benchmarked program is where, instead of increasing the number of branches, we increase the number of times that we introduce recursion. We do this exactly 10 times as well. We want the actual recursion to take place in the scope of the most outer `R`, which means we also have to weaken the scope several times before we can reach the recursion variable.

ST	ST-N	ST-C	ST-(N+C)
19.9 µs	11.1 µs	15.7 µs	6.82 µs

Again, this benchmark clearly benefits from normalizing the program. Furthermore, the combination of normalizing and codensity causes the largest reduction in time.

Now we will perform benchmarking for session typed programs evaluating to Cloud Haskell programs. Unfortunately, we were not able to make use of criterion as these

in all configurations resulted in exceptions or the reported times could not be trusted. The cause of this is that we require starting a node to execute a process. Initially, we would start a node for each individual benchmark test. However, the results then also include the time it took to start and shutdown the node, which would then only dilute the results. Furthermore, we noticed that shutting down a node would not always be completed before the next benchmark test was started. This would then throw an error saying that the port was still in use. Another attempt was to start the node before running the benchmarks. Then a single benchmark test consisted of computing a given program on the already running node. The consequence of this is that the order in which the benchmarks are executed matters. Earlier run benchmarks would then have the best results, almost regardless of what was being tested. Ultimately, we ended up not using criterion and instead we created a benchmarking program that only does a single benchmarking test. We start a node, get the current time, execute the benchmark, again get the 'newer' current time and finally calculate the difference in time. Each different benchmarking test was repeated only 10 times, instead of the 1000 times as done by criterion. The results are therefore less likely to give an accurate representation of the performance. Regardless of that, we found that there was a clear difference in the measured times for each configuration.

For this benchmark we modified the session typed program of the first benchmark results and modified it so that it spawns a session instead. We ran it in three different configurations:

- ST : The same base session typed program as before
- ST-O : Session typed program that has been optimized such that it sends a single integer for branching
- ST-(O+C) : The previous, but now it also uses codensity.

We included codensity, but did not give it a separate configuration, because we had already concluded from our teletype benchmarking that is does indeed improve performance. Likewise, we did not include normalization because the to be normalized program is already in normal form thereby reducing performance.
We compare the above configurations to Cloud Haskell equivalents.

- CH : Cloud Haskell equivalent of the base session typed program
- CH-UM : The same, but uses untyped messaging instead of typed messaging.

CH	CH-UM	ST	ST-O	ST-(O+C)
198 µs	454.4 µs	501 µs	430.2 µs	381.8 µs

The time difference between *CH* and *ST* should make it clear that session typed programs definitely worsen the performance. Though, this should be a given considering the overhead of using a free monad and that we have to transmit booleans over the network when branching. The reduction in speed caused by the latter can be reduced as

is shown by *ST-O*. The branching consisted of only two branches, but was still enough to give a noticable improvement. As has been the case in every benchmark, the use of codensity always further improves performance. The final configuration that we haven't discussed yet is that of *CH-UM*. Instead of using a typed channel like in *CH*, it uses untyped messaging. The consequence is that the performance is severly reduced to that of session typed programs. In section 4.4, we replaced untyped messaging as our underlying method for communication with session typed typed channels. Besides arguing that this was to fix certain problems that could occur in specific situations, it now seems that using typed channels over untyped messaging has a positive effect on the performance of session typed programs.

The results of our benchmarking have shown that using session typed programs certainly does lead to worse performance. This decrease in performance is most likely caused by the overhead of using a free monad to generate an AST and subsequently to evaluate this AST to the `Process` monad. However, there are ways to slightly improve the performance. Specifically, using codensity will always lead to better results. Also, reducing how much communication is required for branching can improve performance. On the other hand, normalizing session typed programs may lead to much better results, but it can also reduce performance. Some of the benchmarks we used were quite extreme in that we knew normalization would probably improve the performance. This really showed in the results, but these benchmarks were not very realistic. If performance is important, one should probably be wary of normalizing their programs.

The benchmarking that we performed is definitely not exhaustive in any way. We were only able to test a limited amount of programs given the amount of time that we had left for writing this thesis. Furthermore, we also did not consider inlining functions such as `eval` or other types of optimizations to force GHC to generate faster code. While it is reasonable to believe that a more exhaustive benchmarking would give a more accurate representation of the performance of the session typed library, we hoped to only give an indication of how the performance of a session typed program would compare to an equivalent Cloud Haskell program.

8 Conclusion

The goal of writing the session types library described in this thesis is to give a programmer the option to add more type safety to his Cloud Haskell programs. It is also important that writing programs using this library shouldn't be much more difficult than writing arbitrary Cloud Haskell programs. We believe that for both goals we partially succeeded.
Using our session types library you can annotate two Cloud Haskell processes with session types and evaluate these as a session. However, a Cloud Haskell program often consists of more than two processes and these processes may have different computation times. As said before, we cannot spawn a session consisting of three or more processes. Furthermore, the processes in a session are generally expected to take about the same time to compute relative to the size of the session type. As we see it the library is therefore best used when within a Cloud Haskell program there is a need to implement a protocol that only requires two processes. In the future we might expand the library to allow for multi-party sessions.
Arguably, the difficult part of writing a session lies in using the branching and recursion session type combinators. The send and receive functions are straightforward and not that different from their Cloud Haskell equivalent, but there exist no Cloud Haskell equivalent for branching and recursion. Especially, recursion can be more difficult to deal with if you have a nesting of recursions, since you will have to take care in how often you do a weakening before recursing. Though, in most programs it is not likely to see more than a nesting of two recursions. Branching itself shouldn't be too difficult. Most likely will branching be used in a case or if expression. You must simply remember to first select a branch in one of the expression branches. Overall, we believe writing sessions is not that more difficult compared to equivalent Cloud Haskell programs.
One large benefit of using our session types library are that different interpreters may be used to evalute a session typed program. This means that the library may also be used in combination with another networking related library that is possibly even more suited to session types. Even for `Session` are these different interpreters very beneficial. So can we use the pure semantics interpreter or the interactive interpreter to test session typed programs in a much easier fashion than you can test Cloud Haskell programs.
We had expected that the performance of our session types library was going to be worse than the Cloud Haskell library. Our own non-exhaustive benchmarking seems to confirm this. Although we can make some optimizations, if performance is critical then the wiser choice to to keep using the Cloud Haskell library.

9 Future work

9.1 Multi-party

We have previously mentioned that our session type library is not capable of multi-party sessions, although Cloud Haskell programs often consist of more than two processes. If we could somehow allow for multi-party sessions, then we can session type the entire Cloud Haskell program instead of only two processes defined within that program. A multi-party session needs a global session type [8] that describes what each process does and to which other process this relates. A global session type describes a partial ordering of actions, where an action consists of any of the session typed actions that we have described in this thesis. A circular ordering of such actions could potentially reintroduce deadlocking. In the case of a session consisting of three processes, where each process sends something to one process and receives something from the other process, it could be possible that one process expects a message from a second process, the second process expects a message from the third process and the third process expects a message from the first process. For all processes to do a send they must first receive a message, but that results in a deadlock. In this case duality is not violated, because no two processes expect messages from each other or send messages to each other, but the cyclic nature of the session type causes the deadlock.
The addition of global session types also calls for a session typed way to send (or receive) messages to different processes. In the work of Imai et al. [9] they allow the creation of session typed channels, different from our own session typed channel, that can be passed to a send or receive. The capability now holds a type-level list of session types instead of a single session type. The variable representing a channel, that can be passed around through the program, denotes a de Bruijn index that can be used to access a session type in the type-level list. This solution can also be used for programs with a global session type to allow communication between different processes. Additionally the channels would also have to be tagged with a process identifier, such that it cannot be used to communicate with different processes.
Two more extensions that can be made to the multi-party session is to allow forking of new processes and sending channels over channels [16].

9.2 Binding

Our current solution for session typed recursion is to delimit the scope of recursion and then to use a recursion variable to denote in which scope recursion should occur. The scope can also be weakened to be able to recurse further back into the program.

However, if we often delimit the scope, we might also often have to expand it again to recurse in a specific scope. This can be annoying for the user, but potentially also worsens performance, because the AST grows everytime you delimit or weaken the scope. In the work of Pucella and Tov they describe a slightly different approach to recursion. They annotate their recursion variables with de Bruijn numbers that denote the scope of the recursion. Other than that their implementation is actually quite similar to ours, but it does give rise to an interesting thought. Can we make use of de Bruijn numbers as indices to our context, such that we merely need to index to the right scope for recursion to occur, without having to weaken the scope as many times as the de Bruijn number indicates?
A different approach is taken for HOAS [1]. They define a type class that exposes a function for each of their terms. One function is `lam`, which is a binder binding a variable. We can possibly take a similar approach, where we replace `lam` with a `rec`. Now `rec` binds a recursion variable. The hoped benefit here is that we would no longer need to do weakening. However, the context of the recursion variable will be set the moment `rec` is used. Then in a nesting of recursion the context of the recursion variable will still have to be weakened. Furthermore, we can define a fixpoint combinator for recursion that essentially does what `rec` does without having to define a type class or define a new term.

9.3 Pure semantics with QuickCheck

In the *Beauty in the beast* paper [21] they show how you can make a pure model of an otherwise impure model and subsequently use the pure model to test their code using QuickCheck [3]. QuickCheck is a tool for automated testing of Haskell programs. QuickCheck tests properties of pure functies by generating random, or in a specified way, input that is then given to the function. The result is then compared to another function that was given the same input. For example, a property of a function that should reverse a list is that reversing the list twice should always equal the original list. The input generated for this test are lists of arbitrary size.
The question is whether we can also use QuickCheck to verify properties of session typed programs. Session typed programs have complex types and the input given to the teletype interpreter must match this type. This makes it difficult for QuickCheck to generate the appropriate input, because it does/can not use the session type for this purpose. A potential solution, in Haskell, would be to write singleton [5] type classes that generates input given some session type. You should be able to pass a session type to the type class that then returns an AST matching that type that can be passed to the teletype interpreter. Done in a dependently typed language, such as Idris [2], we could avoid having to use a type class. Instead, we define a function that takes an implicit argument that is a singleton term. By pattern matching on the singleton term, the type, a session type, is instantiated allowing us to return the appropriate input constructor.

Bibliography

[1] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM, 2009.

[2] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[3] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[4] Edsko de Vries and Andres Löh. True sums of products. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 83–94. ACM, 2014.

[5] Richard A Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2013.

[6] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *ACM SIGPLAN Notices*, volume 46, pages 118–129. ACM, 2011.

[7] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, 2005.

[8] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

[9] Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in haskell. *arXiv preprint arXiv:1110.4163*, 2011.

[10] Oleg Kiselyov. Simple variable-state 'monad'. *Mailing list message, December*, 2006.

[11] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM, 2004.

[12] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[13] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL*, volume 4, pages 56–70. Springer, 2004.

[14] Bruno C d S Oliveira and Andres Löh. Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, pages 87–96. ACM, 2013.

[15] Riccardo Pucella and Jesse A Tov. Haskell session types with (almost) no class. In *ACM Sigplan Notices*, volume 44, pages 25–36. ACM, 2008.

[16] Matthew Sackman and Susan Eisenbach. Session types in haskell: Updating message passing for the 21st century. 2008.

[17] Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. The constrained-monad problem. In *ACM SIGPLAN Notices*, volume 48, pages 287–298. ACM, 2013.

[18] Jan Stolarek, Simon Peyton Jones, and Richard A Eisenberg. Injective type families for haskell. In *ACM SIGPLAN Notices*, volume 50, pages 118–128. ACM, 2015.

[19] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. A unified semantics for future erlang. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, pages 23–32. ACM, 2010.

[20] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4): 423–436, 2008.

[21] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36. ACM, 2007.

[22] Janis Voigtländer. Asymptotic improvement of computations over free monads. In *International Conference on Mathematics of Program Construction*, pages 388–403. Springer, 2008.

[23] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.