Reversible Session-Based Concurrency in Haskell*

Folkert de Vries[1] and Jorge A. Pérez[1 [0000−0002−1452−6180]]

University of Groningen, The Netherlands,

Abstract. Under a reversible semantics, computation steps can be undone. For message-passing, concurrent programs, reversing computation steps is a challenging and delicate task; one typically aims at formal semantics which are *causally-consistent*. Prior work has addressed this challenge in the context of a process model of multiparty protocols (choreographies) following a so-called *monitors-as-memories* approach. In this paper, we describe our ongoing efforts aimed at implementing this operational semantics in Haskell.

Keywords: Reversible computation · Message-passing concurrency · Session Types · Haskell.

* F. de Vries is a BSc student.

1 Introduction

Concurrent systems run processes that communicate over channels. communication introduces a new class of bugs: deadlocks. The simplest example of deadlock is a receive on an empty channel:

```
main = do
    channel <- Channel.new
    -- hangs forever
    message <- Channel.read channel
```

Deadlock errors are difficult to spot and debug. Solutions to prevent common cases of deadlock have been proposed, but haven't really found their way into the mainstream.

Additionally, concurrent systems often perform a list of individual operations that are meant to be one whole. In the snippet below, there is a point where the money is subtracted from `A` but not yet added to `B`. If an error occurs at this point, the system should revert back to the initial state: Just failing silently makes the money disappear.

```
transfer sum = do
    balanceA <- DB.read accountA
    balanceB <- DB.read accountB
    if balanceA >= sum then do
        DB.write accountA (balanceA - sum)
        -- danger zone
        DB.write accountB (balanceB + sum)
    else
        return ()
```

Moving back to the initial state is non-trivial. How can we be sure that we've cleaned up all of our actions and use minimal resources?

Solutions have been proposed that try to solve common deadlock scenarios and reversing evaluation. but it's not made it's way into our languages and libraries. This paper gives an implementation of a framework presented in *reference ppdp* that allows reversing of computation steps and verifies communication.

Our contributions are

- We provide a haskell implementation of a concurrent calculus featuring session types and reversibility
  - We define the **process calculus** in section 2.1 To make actually write example programs pleasant, we will use the `Free` monad to provide convenient syntax

- We define **session types** for the calculus in section 2.3. The session types feature nested recursion and choice.
- We describe the information that needs to be stored for **reversibility** in section 2.6.
- We provide **forward and backward evaluation functions** in section 3.
- We show that the system preserves the properties of the formal definition (section w), notably *causal consistency*

– We describe why purity and reversible computation work well together

2 The Main Idea

The PPDP paper builds a system that features three basic concepts: a process calculus, session types and reversibility. These three ideas are combined to create more robust concurrent programs.

The process calculus is a very low-level programming language that allows concurrent communication. Session types verify the communication to statically eliminate or dynamically detect deadlocks. Reversibility - backwards evaluation of our process calculus - gives the ability to recover from errors or to safely fail.

2.1 process calculi

We must first of all define what concurrent computation is. We use a model called the pi-calculus as our starting point.

The pi-calculus is to concurrent computation much like what the lambda calculus is for sequental computing: A simple model that is convenient for reasoning and constructing proofs. The pi-calculus is defined as

$P, Q, R ::= \overline{x}\langle y\rangle.P$	Send the value y over channel x, then run P
$\mid x(y).P$	Receive on channel x, bind the result to y, then run P
$\mid P \mid Q$	Run P and Q simultaneously
$\mid (\nu x)P$	Create a new channel x and run P
$\mid\ !P$	Repeatedly spawn copies of P
$\mid\ 0$	Terminate the process

Reduction can occur when there is a send and a receive over the same channel in parallel.

$$\overline{x}\langle z\rangle.P \mid x(y).Q \to P|Q[z/y]$$

The PPDP paper makes a few generalizations to this calculus. The calculus used in PPDP is given by

$P, Q ::=$	$u!\langle V\rangle.P$	Send the value V, then run P
	$u?(x).P$	Receive and bind the result to x, then run P
	$u \triangleleft \{l_i.P_i\}_{i\in I}$	Select an option, then broadcast the choice, then run P_i
	$u \triangleright \{l_i : P_i\}_{i\in I}$	Offer options, receive a choice by the selector, then run P_i
	$(P \mid Q)$	Run P and Q simultaneously
	$X \mid \mu X.P$	Variable and function abstraction
	$V\, u$	function application
	$(\nu\, n)$	name restriction: make n unused in a term
	$\mathbf{0}$	terminate

We still have sending, receiving and parallel, and allow recursion and termination. Channel creation has been removed for simplicity: we'll use only one globally available channel. This channel is implemented as a queue which means that sends are non-blocking.

In addition, we introduce. This is where instead of a value, a label is sent from a selector to an offerer. Both parties will pick the branch that the label corresponds to. We'll see why this extension is useful in the next section.

Finally, we allow the sending of thunks - functions that take `Unit` as their argument and return a process term (i.e. a piece of program that can be executed). The fact that we can send programs - and not just values - means that our calculus is a higher-order calculus. The sending of thunks provides "protocol delegation via abstraction passing".

The definition of functions in the PPDP paper is a bit strange: there are two ways of recursion - on the process level and on the value level. In the implementation the former has been removed.

2.2 Implementing the PPDP calculus

The implementation uses an algebraic data type (union type/sum type) to encode all the constructors of the grammar. We use the `Fix` type to factor out recursion from the grammar. As mentioned we omit the process-level recursion.

```haskell
type Participant = String
type Identifier = String

type Program value = Fix (ProgramF value)

data ProgramF value next
    -- communication primitives
    = Send
        { owner :: Participant
        , value :: value
        , continuation :: next
        }
    | Receive
        { owner :: Participant
        , variableName :: Identifier
        , continuation :: next
        }

    -- choice primitives
    | Offer Participant (List (String, next))
    | Select Participant (List (String, value, next))

    -- other constructors
    | Parallel next next
    | Application Identifier value
    | NoOp
    deriving (Functor)
```

We also need values in our language. In the paper they are defined as

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
u, w & ::= & n \mid x, y, z & n, n' & ::= & a, b \mid s_{[p]} \\
v, v' & ::= & \texttt{tt} \mid \texttt{ff} \mid \cdots & & & \\
V, W & ::= & a, b \mid x, y, z \mid v, v' \mid \lambda x.\, P & & &
\end{array}
$$

But to write more interesting examples we've also added some operations on integers and booleans as builtins.

```haskell
data Value
    = VBool Bool
    | VInt Int
```

```
    | VString String
    | VUnit
    | VIntOperator Value IntOperator Value
    | VComparison Value Ordering Value
    | VFunction Identifier (Program Value)
    | VReference Identifier
    | VLabel String
```

2.3 Session Types

As programmers we would like our programs to work as we expect. With concurrent programs, fitting the whole program in one's head becomes increasingly difficult as the application becomes larger.

Data types are a tool to rule out classes of bugs, so the programmer doesn't have to think about them. Type systems range from very loose (dynamic languages like javascript and python) to very restrictive (Coq, Agda, Idris).

In the late 90s, a similar tool was developed for "typing" and checking the interaction between concurrent processes: Session types. Session types provide three key properties:

- **Order** every participant has an ordered list of sends and receives that it must perform
- **Progress** every sent message is eventually received
- **Safety** sender and receiver always agree about the type of the sent value

2.4 Global Types {global-types}

The simplest non-trivial concurrent program has two participants. In this case, the types of the two participants are exactly dual: if the one sends, the other must receive. However, for the multiparty (3 or more) use case, things aren't so simple.

We need to define globally what transactions occur in our protocol.

```
simple :: GlobalType String String
simple = GlobalType.globalType $ do
    transaction "carol" "bob" "Bool"
    transaction "alice" "carol" "int"
```

We can see immediately that **safety and progress are guaranteed**: we cannot construct a valid global type that breaks these guarantees. A more realistic example that we'll use as a running example is the three buyer protocol, defined as follows:

$$
\begin{aligned}
G = \ & \mathtt{A} \to \mathtt{V} : \langle \mathsf{title} \rangle.\ \ \mathtt{V} \to \{\mathtt{A}, \mathtt{B}\} : \langle \mathsf{price} \rangle. \\
& \quad \mathtt{A} \to \mathtt{B} : \langle \mathsf{share} \rangle.\ \ \mathtt{B} \to \{\mathtt{A}, \mathtt{V}\} : \langle \mathsf{OK} \rangle. \\
& \quad\quad \mathtt{B} \to \mathtt{C} : \langle \mathsf{share} \rangle.\ \ \mathtt{B} \to \mathtt{C} : \langle \{\{\diamond\}\} \rangle. \\
& \quad\quad\quad \mathtt{B} \to \mathtt{V} : \langle \mathsf{address} \rangle.\ \ \mathtt{V} \to \mathtt{B} : \langle \mathsf{date} \rangle.\mathtt{end}
\end{aligned}
$$

In this protocol, Alice and Bob communicate with the Vendor about the purchase of some item. Near the end of the protocol, Bob has to leave and transfers the remainder of his protocol to Carol. She will also be sent the code - a **thunk process** $\{\{\diamond\}\}$ - to complete Bob's protocol, and finish the protocol in his name by evaluating the sent thunk.

The full definition of global types in the haskell implementation is given by

```haskell
type GlobalType participant u =
    Free (GlobalTypeF participant u) Void

data GlobalTypeF participant u next
    = Transaction
        { from :: participant
        , to :: participant
        , tipe :: u
        , continuation ::  next
        }
    | Choice
        { from :: participant
        , to :: participant
        , options :: Map String next
        }
    | End
    | RecursionPoint next
    | RecursionVariable
    | Weaken next
    deriving (Functor)
```

We can now see why choice is useful: it allows us to branch on the session type level. For instance, one branch can terminate the protocol, and the other can start from the beginning.

The final three constructors are required for supporting nested recursion. A `RecursionPoint` is a point in the protocol that we can later jump back to. A `RecursionVariable` triggers jumping to a previously encountered `RecursionPoint`. By default it will jump to the closest and most-recently encountered `RecursionPoint`, but `WeakenRecursion` makes it jump one `RecursionPoint` higher, encountering 2 weakens will jump 2 levels higher etc.

Using `Monad.Free`, we can write the ThreeBuyer example type as

```
data MyParticipants = A | B | C | V
    deriving (Show, Eq, Ord, Enum, Bounded)

data MyType = Title | Price | Share | Ok | Thunk | Address | Date
    deriving (Show, Eq, Ord)

globalType :: GlobalType.GlobalType MyParticipants MyType
globalType = GlobalType.globalType $ do
    GlobalType.transaction A V Title
    GlobalType.transactions V [A, B] Price
    GlobalType.transaction A B Share
    GlobalType.transactions B [A, V] Ok
    GlobalType.transaction B C Share
    GlobalType.transaction B C Thunk
    GlobalType.transaction B V Address
    GlobalType.transaction V B Date
    GlobalType.end
```

2.5 Local Types

A global type can contain inherent parallelism: the order of its steps is not fully defined. Checking for correctness against a global type is therefore quite hard. The solution is to project the global type onto its participants, creating a local type.

The local type is an ordered list of steps that the participant must execute to comply with the protocol. Thus the third property **Order** is satisfied for local types.

The projection is mostly straightforward, except for choice. Because we allow recursion, a branch of a choice may recurse back to the beginning. When this occurs, all participants have to jump back to the beginning, so every choice must be communicated to all participants.

```
-- some latex snipped with the full projection rules
```

2.6 reversiblilty

The third component of the system is reversibility. The idea here is that we can move back to previous program states, reversing forward steps.

Reversibility is useful in a concurrent setting because many concurrent systems are transactional: there is a list of steps that we want to treat as an atomic action.

That means that either we succeed, or we want to not have done anything at all. So if we fail in the middle of the steps, we can't just abort: we need to put the system back into its initial state.

The naive way to achieve reversal is to store a snapshot of the initial state. The memory consumption of this method is a deal-breaker. For instance, the system may interact with a large database. Keeping many copies of the database around is inconvenient and may not even be physically possible.

A nicer approach is to keep track of our forward steps, and store just enough information to reverse those steps. Conceptually, we're leaving behind a trail of breadcrumbs so we can always find our way back.

The challenge, then, is to find the minimal amount of information that we need to store for every instruction. Broadly, we need to track information about two things: the type and the process.

For the type we define a new data type called `TypeContext`. It contains the actions that have been performed and for some stores a bit of extra information like the `owner`.

On the process level there are four things that we need to track:

- used variable names in receives

The rest of the program depends on the name that we assign to a received value. Therefore we need to restore the original name when we revert. Used variable names are stored on a stack that is popped when we revert a receive.

```
program = do
    decision <- H.receive
    H.send decision
    H.send decision
```

- unused branches

When a choice is made and then reverted, we want all our options to be available again. The current semantics aren't able to make a different choice, but it will be in the future.

```
type Zipper a = (List a, a, List a)

data OtherOptions
    = OtherSelections (Zipper (String, Value, Program Value))
    | OtherOffers (Zipper (String, Program Value))
```

We need to store the selection that has been made. The zipper contains the taken path as its central `a`, and the options before and after as the lists on either side.

– function applications

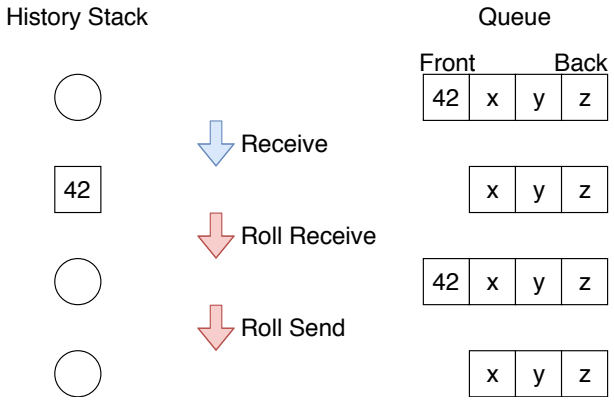
When applying a function we lose information about the applied function and the argument. Therefore we store them in a map and associate them with a unique identifier. This identifier is given to the type-level "breadcrumb".

Q: Why is this? why not a stack

```
Map Identifier (Value, Value)
```

– messages on the channel

When a value is sent, the sender loses track of what has been sent. Therefore, reverting a send/receive pair must move the value from the receiver via the queue back to the sender. To this end, the queue holds on to received values: the value is popped from the queue but pushed onto the history stack. Now when the receive is rolled, the value is moved back onto the queue, and when the send is rolled the value is moved from the head of the queue into the sender's process.



Q: Why store the information in the queue stack, and not move the value from the receiver's variable definitions onto the top of the queue?

2.7 Abstraction passing is protocol delegation

Effectively this means that instructuions need to keep track of whose protocol they should move forward. The `owner` field stores this information.

Q: why/when is delegation actually useful in practice? possibly we can move part of the protocol away from a location that is terminated?

2.8 Properties of reversibility

The main property that reversing needs to preserve is *causal consistency*: the state that we reach when moving backward is a state that could have been reached by moving forward only.

3 Putting it all together

With all the definitions encoded, we can now define forward and backward evaluation of our system. Our aim is to implement

```
type Session a = StateT ExecutionState (Except Error) a

forward  :: Location -> Session ()
backward :: Location -> Session ()
```

Where

- `ExecutionState` stores the state of the world
- `Error` can be thrown when something fails
- `Location` defines places where code is ran concurrently (threads or machines)

3.0.0.1 Type checking We store all the information about a participant in a type called `Monitor`.

```
data Monitor value tipe =
    Monitor
        { _localType :: LocalTypeState tipe
        , _recursiveVariableNumber :: Int
        , _recursionPoints :: List (LocalType tipe)
        , _store :: Map Identifier value
        , _usedVariables :: List Binding
        , _applicationHistory :: Map Identifier (value, value)
        }
        deriving (Show, Eq)

data Binding = Binding { _visibleName :: Identifier, _internalName :: Identifier } deriving
```

The `TypeContext` and `LocalType` are stored as a tuple. Really, this gives a curser into the local type, where everything to the left is the past and everything to the right is the future.

The next two fields are for keeping track of recursion in the local type. the `recursiveVariableNumber` is an index into the `recursionPoints` list: when a `RecursionVariable` is encountered we look at that index to find the new future local type.

Then follow two fields used for reversibility: the stack of used variable names and the store of function applications. Finally there is a variable store with the currently defined bindings.

With this data structure in place, we can define `ExecutionState`. It contains some counters for generating unique variable names, a monitor for every participant and a program for every location. Additionally every location has a default participant and a stack for unchosen branches.

```
data ExecutionState value =
    ExecutionState
        { variableCount :: Int
        , locationCount :: Int
        , applicationCount :: Int
        , participants :: Map Participant (Monitor value String)
        , locations :: Map Location (Participant, List OtherOptions, Program value)
        , queue :: Queue value
        , isFunction :: value -> Maybe (Identifier, Program value)
        }
```

The message queue is global and thus also lives in the `ExecutionState`. Finally we need a way of peeking into values, to see whether they are functions and if so, to extract their bodies for application.

4 Convenient syntax with the free monad

The types we've defined for `Program`, `GlobalType` and `LocalType` form recursive tree structures. Because they are all new types, there is no easy way to traverse them. A common idiom is to factor out recursion using `Data.Fix` (see section 8.6).

Even with `Fix`, the expressions are tedious to write. For `LocalType` that doesn't matter because it is (almost) never written by hand. For `Program` and `GlobalType` - which we'll construct by hand a lot - we want nicer syntax. To this end we can use the free monad (see section 8.8). The free monad is a standard method in haskell to construct domain-specific languages. An extra bonus is that we can use do-notation (section 8.2) to write our examples.

Concretely, the free monad makes it possible to write

```
program = compile "Alice" $ do
    result <- receive
    send (VInt 42)
```

instead of

```
program =
    Receive
        { owner = "Alice", variableName = "result", continuation =
            Send
                { owner = "Alice", value = VInt 42, continuation = NoOp }
        }
```

By using `StateT` (see 8.5) we can thread through the participant name and generate unique variable names using a counter. `Free` makes it possible build up and produce a `Program` at the end.

```
newtype HighLevelProgram a =
    HighLevelProgram (StateT (Participant, Int) (Free (ProgramF Value)) a)
    deriving (Functor, Applicative, Monad, MonadState (Participant, Int))

send :: Value -> HighLevelProgram ()
send value = do
    (participant, _) <- State.get
    HighLevelProgram $ liftF (Send participant value ())

terminate :: HighLevelProgram a
terminate = HighLevelProgram (liftF NoOp)

-- similar for receive, select, etc.
```

Here we use the unit type `()` as a placeholder or hole. To ensure that the program is well-formed and doesn't contain any holes when we evaluate it, all branches must end with `terminate`. We use the fact that `terminate :: HighLevelProgram a` contains a free type variable `a` which can unify with `Void`, the type with no values. Thus `Free f Void` can contain no `Pure` because the `Pure` constructor needs a value of type `Void`, which don't exist. For more information see the appendix section 8.9.

5 Benefits of pure functional programming

It has consistently been the case that sticking closer to the formal model gives better code. The abilities that Haskell gives for directly specifying formal statements is invaluable. The main killer feature is algebraic data types (ADTs) also known as tagged unions or sum types.

Observe the formal definition and the haskell data type for global types.

$$G, G' \quad ::= \quad \mathtt{p} \to \mathtt{q} : \langle U \rangle . G \;\Big|\; \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \;\Big|\; \mu X . G \;\Big|\; X \;\Big|\; \mathtt{end}$$

```haskell
data GlobalTypeF u next =
    Transaction {..} | Choice {..}  | R next | V | End | Wk next
```

They correspond directly. Moreover, we know that these are all the ways to construct a value of type `GlobalTypeF` and can exhaustively match on all the cases. Functional languages have had these features for a very long time. In recent years they have also made their way into non-functional languages (Rust, Swift, Kotlin).

5.1 Reversibility and Purity

Secondly, purity and immutability are very valuable in implementing and testing reversibility. The type system can actually guarantee that we've not forgotten to revert anything.

In a pure language, given functions `f :: a -> b` and `g :: b -> a` to prove that f and g are inverses it is enough to prove that `f . g = identity && g . f = identity`. In an impure language, we also have to consider the outside world, some context C.

`f (C[x]) = C'[y] => g (C'[y]) = C[x]`

Because we don't need to consider a context here, checking that reversibility works is as simple as comparing initial and final states.

6 Discussion

7 Conclusion

we've given an encoding of the PPDP semantics in the haskell programming language

8 Notes on haskell notation and syntax

8.1 Performing IO in Haskell

One of Haskell's key characteristics is that it is pure. This means that our computations can't have any observable effect to the outside world. Purity enables us to reason about our programs (referential transparency) and enables compiler optimizations.

But we use computres to solve problems, and we want to be able to observe the solution to our problems. Pure programs cannot produce observable results: The computer becomes very hot but we can't see our solutions.

So we perform a trick: we say that constructing our program is completely pure, but evaluating may produce side-effects like printing to the console or writing to a file. To separate this possily effectful code from pure code we use the type system: side-effects are wrapped in the `IO` type.

```
-- print a string to the console
putStrLn :: String -> IO ()

-- read a line of input from the console
readLn :: IO String
```

A consequence of having no observable effects is that the compiler can reorder our code for faster execution (for instance by minimizing cache misses). But this will wreak havoc when performing IO: we want our IO actions to absolutely be ordered.

The trick we can pull here is to wrap the later actions in a function taking one argument, and piping the result of the first action into that function. The result is only available when the first action is done, so the first action is always performed before the rest: we've established a data-dependency between the first and the remaining actions that enforces the order.

The piping is done by the `>>=` operator. In the haskell literature this function is refered to as `bind`, but I think the elm name `andThen` is more intuitive (at least for `IO`). A program that first reads a line and then prints it again can be written as

```
main =
    readLn >>= (\line -> putStrLn line)

andThen :: IO a -> (a -> IO b) -> IO b
andThen = (>>=)

main2 = do
    readLn `andThen` (\line -> putStrLn line)
```

The `putStrLn` can only be evaluated when `line` is available, so after `readLn` is done. More technically the `(>>=) :: IO a -> (a -> IO b) -> IO b` operator will first evaluate its first argument `IO a`, in this case `IO String` (that string is the line we have read). Then it "unwraps" that `IO String` to `String` to give it as an argument to `a -> IO b` (here `String -> IO ()`). Note that we can never (safely) go from `IO a -> a`. The unwrapping here is only valid because the final return type is still `IO something`.

When printing two lines, we can use a similar trick to force the order

```
main =
    putStrLn "hello " >>= (\_ -> putStrLn "world")
```

Here we ignore the result of the first `putStrLn`, but the second `putStrLn` still depends on its return value. Thus it has to wait for the first `putStrLn` to finish before it can start.

8.2 Do-notation

Writing nested functions in this way quickly becomes tedious. That is why special syntax is available: do-notation

```
main1 = do
    line <- readLn
    putStrLn line

main2 = do
    putStrLn "hello "
    putStrLn "world"
```

do-notation is only syntactic sugar: it is translated by the compiler to the nested functions that we have seen above. The syntax is very convenient however. Additionally, we can use it for all types that implement `>>=`: all instances of the `Monad` typeclass.

8.3 Monads

`Monad` is a haskell typeclass (and a concept from a branch of mathematics called category theory). Typeclasses are sets of types that implement some functions, similar to interfaces or traits in other languages.

The monad typeclass defines two methods: bind/andThen/`>>=` which we've seen and `return :: Monad m => a -> m a`. The `Monad m =>` part of the signature constrains the function to only work on types that have a `Monad` instance.

I hope the above already gives some intuition about monad's main operator `>>=`: it forces order of evaluation. A second property is that `Monad` can merge contexts with `join :: Monad m => m (m a) -> m a`. A common example of join is `List.concat :: List (List a) -> List a`. A bit more illustrative is the implementation for `Maybe`

```
join :: Maybe (Maybe a) -> Maybe a
join value =
    case value of
        Nothing ->
            Nothing

        Just (Nothing) ->
            Nothing

        Just (Just x) ->
            Just x
```

If the outer context has failed (is `Nothing`), then the total computation has failed and there is no value of type `a` to give back. If the outer computation succeeded but the inner one failed, there is still no `a` and the only thing we can return is `Nothing`. Only if both the inner and the outer computations have succeeded can we give a result back.

In short

- `Monad` is a haskell typeclass
- it has two main functions
  - bind or andThen or `>>= :: Monad m => m a -> (a -> m b) -> m b`
  - `return :: Monad m => a -> m a`
- `Monad` forces the order of operations and can flatten wrappers
- `Monad` allows us to use do-notation
- `IO` is an instance of `Monad`

Much material on the web about monads is about establishing the general idea, but really the exact meaning of `>>=` can be very different for every instance. Next we'll look at some of the types used in the code for this thesis.

8.4 Except

Except is very similar to `Either`:

```
data Either a b
    = Left a
    | Right b
```

We use this type to throw and track errors in a pure way.

```
throwError :: e -> Except e a

data Error
    = QueueEmpty
    | ...

popQueue :: List a -> Except Error (a, List a)
popQueue queue =
    case queue of
        [] ->
            Except.throwError QueueEmpty

        x : xs ->
            return ( x, xs )
```

Except and Either have a `Monad` instance. In this context `return` means a non-error value, and `>>=` allows us to chain multiple operations that can fail, stopping when an error occurs.

8.5 State and StateT

`State` is a wrapper around a function of type `s -> (a, s)`

```
newtype State s a = State { unState :: s -> (a, s) }
```

It is used to give the illusion of mutible state, while remaining completely pure. Intuitively, we can compose functions of this kind.

```
f :: s -> (a,   s)
g ::       a -> s -> (b, s)
-- implies
h :: s ->            (b, s)
```

And this is exactly what monadic bind for state does.

```
andThen :: State s a -> (a -> State s b) -> State s b
andThen (State first) tagger =
    State $ \s ->
        let (value, newState) = first s
            State second = tagger value
        in
```

```
            second newState

new :: a -> State s a
new value = State (\s -> (value, s))

instance Monad (State s) where
    (>>=) = andThen
    return = new
```

When we want to combine monads, for instance to have both state and error reporting, we must use monad transformers. The transformer is needed because monads don't naturally combine: `m1 (m2 a)` may not have a law-abiding monad instance.

```
newtype StateT s m a = StateT { runStateT :: s -> m (a, s) }

instance MonadTrans (StateT s) where
    lift :: (Monad m) => m a -> StateT s m a
    lift m = StateT $ \s -> do
        a <- m
        return (a, s)

instance (Monad m) => Monad (StateT s m) where
    return a = StateT $ \s -> return (a, s)
```

The `MonadTrans` typeclass defines the `lift` function that wraps a monadic value into the transformer. Next we define an instance because we can say "given a monad `m`, `StateT s m` is a law-abiding monad.

8.6 Factoring out recursion

A commonly used idiom in our code is to factor out recursion from a data structure, using the `Fix` and `Monad.Free` types. Both require the data type to be an instance of `Functor`: The type is of the shape `f a` - like `List a` or `Maybe a`, and there exists a mapping function `fmap :: (a -> b) -> (f a -> f b)`.

Fix requires the data type to have a natural leaf: a constructor that does not contain an `a`. `Free` on the other hand lets us choose some other type for the leaves.

8.7 Fix

The `Fix` data type is the fixed point type.

```haskell
data Fix f = Fix (f (Fix f))
```

It allows us to express a type of the shape `f (f (f (f (..))))` concisely. For the values of this type to be finite, the `f` must have a constructor that does not recurse to be a leaf. Take for instance this simple expression language

```haskell
data Expr
    = Literal Int
    | Add Expr Expr
```

`Literal` is the only constructor that can occur as a leaf, and `Add` is the only node. Using `Fix` We can equivalently write

```haskell
data ExprF next
    = Literal Int
    | Add next next

simple :: Fix ExprF
simple = Fix (Literal 42)

complex :: Fix ExprF
complex =
    Fix (Add (Fix (Literal 40)) (Fix (Literal 2)))
```

By decoupling the recursion from the content, we can write functions that deal with only one level of the tree and apply them to the full tree. For instance evaluation of the above expression can be written as

```haskell
evaluate :: Fix ExprF -> Int
evaluate =
    Fix.cata $ \expr ->
        case expr of
            Literal v ->
                v

            Add a b ->
                a + b
```

The `cata` function - a catamorphism also known as a fold or reduce - applies evaluate from the bottom up. In the code we write, we only need to make local decisions and don't have to write the plumbing to get the recursion right.

8.8 Monad.Free

The Free monad is very similar to `Fix`, but allows us to use a different type for the leaves, and enables us to use do-notation. Writing expressions with `Fix` can be quite messy, free monads allow us to write examples much more succinctly.

Free is defined as

```
data Free f a
    = Pure a
    | Free (f (Free f a))
```

and as the name suggests `Monad.Free` has a Monad instance.

This then makes it possible to define a functor that represents instructions, define some helpers and then use do-notation to write our actual programs.

```
data StackF a next
    = Push a next
    | Pop (a -> next)
    | End
    deriving (Functor)

type Stack a = Free StackF a

program :: Stack Int
program = do
    push 5
    push 4
    a <- pop
    b <- pop
    push (a + b)

push :: a -> Stack a ()
push v = liftFree (Push v ())

pop :: Stack a a
pop = liftFree (Pop identity)
```

8.9 Guaranteeing well-formedness of Free

We use the free monad to build up programs and global types. A problem with the free monad is that the built-up tree can still contain "holes" because of the `Pure _` branch of `Free`. That is fine while constructing the tree, but when

evaluating it we want all `Pure`s to be gone. There are two ways of enforcing this constraint using the type systems.

We observe that only our leaves have a free type variable. For instance for `HighLevelProgram`, only `terminate` (via NoOp) can have type `HighLevelProgram a` where the `a` is unbound. That means that `terminate` will unify with anything: `HighLevelProgram String`, `HighLevelProgram Int`, etc.

```
terminate :: HighLevelProgram a
terminate = HighLevelProgram (liftF NoOp)
```

There are now two ways forward:

Solution 1: Rank2Types and Universal Quantification

Now if we enforce that our whole program unifies with anything, that implies that all `Pure`s are gone from the structure. Normally, type signatures are valid if there is at least one valid unification for every type variable (i.e. existential quantification). But with the language extension `ExplicitForAll` we can mark type variables as universally quantified: they need to unify with all types. In this case we also need `Rank2Types` because of the position where we want to use `forall`.

```
{-# LANGUAGE Rank2Types #-}
{-# LANGUAGE ExplicitForAll #-}

compile :: Participant -> (forall a. HighLevelProgram a) -> Program Value
compile participant (HighLevelProgram program) = ...
```

Solution 2: Data.Void

The second solution is to use `Data.Void`. `Void` is the data type with zero values, which means there is no valid way of creating a value of type `Void`. Thus a `Free f Void` cannot have any `Pure`s, because they need a value of type `Void` and there are none.

```
import Data.Void (Void)

compile :: Participant -> HighLevelProgram Void -> Program Value
compile participant (HighLevelProgram program) = ...
```

Tradeoffs

In the codebase we went with solution 2 because it produces clearer error messages and doesn't introduce extra language extensions to the project.