Reversible Session-Based Concurrency in Haskell

Folkert de Vries and Jorge A. Pérez

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 efforts aimed at implementing this operational semantics in Haskell. We provide an encoding based on the formal definition of prior work's three core concepts: a process calculus, multiparty session types and a reversible semantics. Additionally, we provide an implementation of the forward and backward transition functions, which make it possible to run and typecheck programs, and finally reflect on uses of the implementation of these semantics.

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

1 Introduction

This paper describes a Haskell implementation of a *reversible semantics* for message-passing concurrent programs. Our implementation is framed within a prolific line of research, which aims at establishing semantic foundations for reversible computing in a concurrent setting (see, e.g., the survey [4]). Our key interest is the interplay of reversibility and message-passing concurrency, which is typically governed by *protocols* among possibly distributed partners. A central observation here is that those protocols may inform the implementation of a reversible semantics.

In a language with a reversible semantics, computation steps can be undone. Thus, a program can perform standard *forward* steps, but also *backward* steps. Reversing a sequential program is not hard: it suffices to record enough information about forward steps in case we wish to return to a prior state using a backward step. Reversing a concurrent program is more difficult: since control may simultaneously reside in more than one point, we require carefully designed *memories* that not only record information about the steps performed in each thread, but also about the *causal dependencies* between steps from different threads. This motivates the definition of reversible semantics which are *causally consistent*, i.e., that ensure that backward steps lead to states that could been

have reached by performing forward steps only [4]. Hence, a causally consistent semantics never leads to states that are not reachable through forward steps.

Causal consistency is an important correctness criterion in the definition of reversible programming languages. The quest for causally consistent reversible semantics for (message-passing) concurrency has led to a number of valuable proposals (cf. [5] and references therein). One common shortcoming in several prior works is that the memories used are rather heavy, and so the resulting reversible semantics are overly complex. This is a particularly notorious limitation in the work of Mezzina and Pérez in [5], which addresses reversibility in the context of concurrent processes which exchange messages following *choreographies*, as defined by *multiparty session types* [2]. The reversible semantics developed in [5] is causally consistent; however, it is unclear whether it can be implemented as actual tools for the analysis of message-passing, concurrent programs.

In this paper, we describe a Haskell implementation of the reversible semantics proposed in [5]. More precisely, we present a Haskell interpreter of message-passing programs written in the reversible process framework in [5]. This allows us to assess in practice the benefits and features of the memories and mechanisms deployed in [5] to enforce causally consistent reversibility. In this process, we have found the use of a functional programming language—and in particular, of Haskell—a natural choice. Haskell has a strong history in language design. Its type system and mathematical nature allow us to faithfully implement the formal semantics and trust that our implementation is correct. In particular, algebraic data types (sums and products) are invaluable in expressing grammars and recursive data structures.

2 The Process Model

3 Introduction

This thesis describes an implementation of theoretical work in the area of concurrent programming. In particular it concerns the use of formal tools to check the correctness of message-passing programs and allow for flexible error handling.

An example of a class of errors in message-passing are **deadlocks**. Consider this example of a receive on an empty channel:

```
main = do
    channel <- Channel.new
    -- will block forever
    message <- Channel.receive channel
```

This snippet uses Haskell's do-notation. the `<-` *is used to bind the result of an effectful computation - like receiving from a channel. See also Appendix A.3*

No value will ever be sent on the channel, so the receive blocks forever: it cannot make any progress.

Failing gracefully is challenging in a concurrent context: the state of the application is typically spread over multiple locations which makes it hard to determine where the error originates and how to compensate for it.

Consider the following example: if Bob fails in receiving a message or loses a message, he will have to repeat the `receive`, but also inform Alice to send the message again.

```
alice channel = do
    Channel.send channel "hello, my name is"
    Channel.send channel "Alice"
    response <- Channel.receive channel

bob channel =
    _ <- Channel.receive channel
    name <- Channel.receive channel
    Channel.send ("Hello, " ++ name)
```

Moving back to the initial state is non-trivial. How can we be sure that we have reverted all our actions? What is the minimal amount of information (and memory) we need for a valid reversal?

The framework presented by Mezinna & Pérez (PPDP'17) [**?**] (from here on referenced as "the PPDP'17 paper") sets out to address both these issues. The paper develops a theoretical approach, using techniques from programming languages and concurrency. However, it does not consider how the presented approach can be implemented in an actual programming language. In earlier work during a short programming project we have looked at combining the CaReDeb debugger [**?**] with the PPDP'17 paper. Here we go further and look at how well the ideas presented in the PPDP'17 paper translate to a programming context, and whether the resulting system turns out to be convenient to use.

In answering these questions this bachelor's thesis makes the following contributions:

1. We begin by analyzing the three core components of the PPDP'17 paper: A process calculus, multiparty session types [1,3] and a reversible semantics. At the same time we give the encoding of the formal definitions of these components into Haskell data types.
   - First, we define and implement a **process calculus** from its foundations. Later, we use the `Free` monad to provide convenient syntax.
   - Next, we look at **multiparty session types** for the implemented calculus. The session types feature nested recursion and labelled choice.
   - Finally, we consider **a reversible semantics** given the two previous concepts. In particular, we look at the information that needs to be preserved when moving forward in order maintain causal consistency.

The rest of the document is structured as follows: Section 4 introduces the three key foundations from the PPDP'17 paper and how they can be encoded into Haskell data types. Section 5 shows how the data types can be used to implement

forward and backward operational semantics and provide the other mechanics needed to evaluate the program. Section 6 provides a more convenient syntax for writing (reversible) message-passing programs. Appendices A.2 through A.11 give background and extra examples for the Haskell concepts and terminology used in the paper.

4 The Main Idea

The PPDP'17 paper builds a programming model that features three basic concepts: a process calculus, multiparty session types and a reversible semantics. The result of combining these ideas allows runtime verification of programs, with the possibility of dealing with failure in a flexible way.

The process calculus is a low-level programming language that allows concurrent communication. The multiparty session types defined in the PPDP'17 paper dynamically validate the communication a process performs against a protocol. Finally, a reversible semantics for the process calculus in combination with local types derived from multiparty session types make it possible to reverse the evaluation of a program. Reversing gives the ability to recover from errors or to safely fail.

In this section we show how these three concepts are formally defined in the PPDP'17 paper and how they are implemented in our Haskell implementation.

4.1 Process Calculi

As already mentioned, a process calculus is a minimal programming language that can model concurrent communication. We will start by considering a well-understood existing calculus: the π-calculus. Next we will look at how the PPDP'17 paper extends it and finally give an implementation as a Haskell data type.

The π-calculus is to concurrent computation much like what the λ-calculus is for sequential computing: a simple model that is convenient for reasoning about a program's correctness. The syntax of the π-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 local to P and run P
$\mid\ !P$	Repeatedly spawn copies of P
$\mid\ 0$	Terminate the process
$\mid\ P + Q$	Non-deterministic choice between P and Q

The relevant part of the operational semantics for this paper is the reduction of send and receive: a π-calculus term can reduce when there is a send and a receive over the same channel running in parallel:

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

With this syntax we can construct programs like so:

$$(\nu x)(\overline{x}\langle 42\rangle.0 \mid x(y).0)$$

This program can be read as "create a channel x and then in parallel send 42 over x then terminate, and receive a value on x and bind it to the name y then terminate." The π-calculus is much too low-level to write real programs in, but it can compute any computable function (because the λ-calculus can be encoded into it [**?**]).

The PPDP'17 paper makes a few generalizations to this calculus. Their calculus is given by the syntax in Fig. 1.

$P, Q ::=$	$u!\langle V\rangle.P$	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
	$(\nu\, n)P$	name restriction: make n local in a term
	$\mathbf{0}$	terminate

Fig. 1. Syntax of Processes

The PPDP'17 paper preserves sending, receiving and parallel, and allow recursion and termination. Channel creation is still part of the grammar (the n in $(\nu\, n)P$ can take a session name $s_{[p]}$) but is restricted to mean that all endpoints in P are simultaneously bound. In practice this means that there is only one globally available message channel. This global channel is implemented as a queue which means that sends are non-blocking.

The primary extension is choice: where the π-calculus has non-deterministic choice between π-terms, the process calculus has labeled deterministic choice. The choice is determined by the state of the program. The selector picks a label and sends it to the offerer. Both parties will coordinate to pick the branch that the label corresponds to and discard the alternative branches. Choice could be

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

Fig. 2. Values: Names a, b, c (resp. s, s') range over shared (resp. session) names. We use session names indexed by participants, denoted $s_{[p]}, s_{[q]}$. Names n, m are session or shared names. First-order values v, v' include base values and constants. Variables are denoted by x, y and recursive variables are denoted by X, Y. The syntax of values V includes shared names, first-order values, but also name abstractions (higher-order values) $\lambda x.P$, where P is a process.

implemented as a combination of send/receive and if-then-else, but they are primitives here because each branch can have a different type (this will become clearer in Section 4.3).

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 (see Section 4.5).

A side-effect of the function values is that there are two ways of defining recursion: using the $\mu X.P$ process constructor or using the $\lambda x.P$ value constructor. In the implementation the former has been removed.

4.2 Implementing the PPDP'17 calculus in Haskell

The implementation uses an algebraic data type (union type/sum type) to encode all the constructors of the grammar. We use the `Fix` type (Appendix A.8) to factor out recursion from the grammar. `Fix` is the fixed point type:

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

`Fix` allows us to concisely express an infinite nesting of a type (of the shape `f (f (f (f (..)))))`. Such a type will look like a tree. For the values of this type to be finite, the `f` must have a leaf constructor. Take for instance this simple expression language

```
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
```

In the above snippet, `next` is a placeholder or hole for an arbitrarily deep nesting of `ExprF`s. The main usage of `Fix` is that it gives traversals and folds of our syntax tree for free because these are generically defined on `Fix`.

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

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

The above snippet shows that defining values of type `Fix ExprF` requires wrapping of constructors with `Fix`. We will describe a better way of writing `Fix`ed values in Section 6.

The full definition of `Program` is given below. 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. To write more interesting examples we extend the types of values that can be used from names, booleans and functions to also include references, integers, strings, and basic integer and logic operators.

```
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
```

We can now write the send/receive example from the Introduction as:

```
example =
    Fix
        ( Parallel
            (Fix (Send "me" (VInt 42) (Fix NoOp)))
            (Fix (Receive "you" "y" (Fix NoOp)))
        )
```

While the above is easily manipulatable with pattern matching and functions from the `Fix` module, it is very hard to write clear examples in this style. Longer expressions need ever more parentheses and in general there is too much syntactical clutter. We will get back to designing a better syntax for defining programs in Section 6.

4.3 Multiparty 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 and program behavior becomes harder to predict.

Most programmers are familiar with data types. Compilers use data types to typecheck programs. Programs that typecheck are statically guaranteed to not have certain classes of bugs. 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 abstract protocols to enforce three key properties:

– **Order:** every protocol participant has a sequence of sends and receives that it must perform;
– **Progress:** every sent message is eventually received;
– **Communication Safety:** sender and receiver always agree about the type of the exchanged values

Next we will look at how these types are defined and how we can check that our programs implement their session type.

Global Types The simplest non-trivial concurrent program has two participants. In this case, the types of the two participants should be exactly dual: if the one sends, the other must receive. However, for the multiparty use case (three or more participants), things are not so simple as duality is hard to define.

We need to define globally what communications occur in our protocol. The snippet below is a basic global type where Carol and Bob first exchange a value of type `Bool`, next Alice and Carol exchange a value of type `Int` and finally the protocol ends:

```
simple :: GlobalType String String
simple = do
    message "carol" "bob" "Bool"
    message "alice" "carol" "Int"
    end
```

We can see immediately that **safety and progress are guaranteed**: Every communication has a send and receive, and they both must agree on the type. We cannot construct a valid global type that does not check for these properties. As a result, every program that implements a global type also has these properties.

A more realistic scenario that we will use as a running example is the three buyer protocol from the PPDP'17 paper. In this protocol, Alice and Bob communicate with the Vendor about the purchase of some item:

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

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** denoted $\{\{\diamond\}\}$ - 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

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

```haskell
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 and taken from [6]. 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` (Appendix A.9), we can write examples with a more idiomatic Haskell syntax.

The snippet below shows the use of nested recursion. There is an outer loop that will perform a piece of protocol or end, and an inner loop that sends messages from `A` to `B`. When the inner loop is done, control flow returns to the outer loop.

```haskell
import GlobalType as G

G.recurse $ -- recursion point 1
    G.oneOf A B
        [ ("loop"
          , G.recurse $ -- recursion point 2
                G.oneOf A B
                    [ ("continueLoop", do
                        G.message A B "date"
                        -- jumps to recursion point 2
                        G.recursionVariable
                      )
```

```
                    , ("endInnerLoop", do
                        -- jumps to recursion point 1
                        G.weakenRecursion G.recursionVariable
                      )
                    ]
          )
        , ("end", G.end)
        ]
```

Similarly, the three buyer example can be written as:

```
-- a data type representing the participants
data MyParticipants = A | B | C | V
    deriving (Show, Eq, Ord, Enum, Bounded)

-- a data type representing the used types
data MyType = Title | Price | Share | Ok | Thunk | Address | Date
    deriving (Show, Eq, Ord)

-- a description of the protocol
globalType :: GlobalType.GlobalType MyParticipants MyType
globalType = do
    message A V Title
    messages V [A, B] Price
    message A B Share
    messages B [A, V] Ok
    message B C Share
    message B C Thunk
    message B V Address
    message V B Date
    end
```

Local Types A global type can contain inherent parallelism: the order of its steps is not fully defined. For instance in `messages V [A, B] Price`, `V` can send the price first to `B` or first to `A`: both are valid according to the type. This parallelism makes checking for correctness against a global type difficult and also unpractical because the global type mentions all participants. The solution is to project the global type onto its participants, creating a local type.

The local type is an ordered list of communication actions that the participant must execute to comply with the protocol. Thus the **order** property is enforced 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.

The PPDP'17 paper and many other papers deal with this problem by disallowing different branches using different participants. In practice this means that all branches must perform the same communication to satisfy the global type, even if the communication is not needed in every branch (some branches may have dummy communication).

In the Haskell implementation we use a different method where every choice causes a broadcast of that choice to the other participants. Once again, some participants may not need the information about the choice and the communication with them is just to satisfy the type checker.

The underlying is that problem global types only have meaning with respect to their projection into local types. If a global type cannot be projected - even though it is grammatically valid - it has no meaning. A simple projection disallows many (grammatically valid) global types and implementations. A more sophisticated projection allows more global types, but with the associated cost of more dummy communications.

$$(\mathtt{p} \to \mathtt{q} : \langle U \rangle.G)\!\downarrow_{\mathtt{r}} = \begin{cases} \mathtt{q}!\langle U \rangle.(G\!\downarrow_{\mathtt{r}}) & \text{if } \mathtt{r} = \mathtt{p} \\ \mathtt{p}?\langle U \rangle.(G\!\downarrow_{\mathtt{r}}) & \text{if } \mathtt{r} = \mathtt{q} \\ (G\!\downarrow_{\mathtt{r}}) & \text{if } \mathtt{r} \neq \mathtt{q}, \mathtt{r} \neq \mathtt{p} \end{cases}$$

$$(\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I})\!\downarrow_{\mathtt{r}} = \begin{cases} \mathtt{q} \oplus \{l_i : (G_i\!\downarrow_{\mathtt{r}})\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{p} \\ \mathtt{p} \& \{l_i : G_i\!\downarrow_{\mathtt{r}}\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{q} \\ (G_1\!\downarrow_{\mathtt{r}}) & \text{if } \mathtt{r} \neq \mathtt{q}, \mathtt{r} \neq \mathtt{p} \text{ and} \\ & \forall i, j \in I. G_i\!\downarrow_{\mathtt{r}} = G_j\!\downarrow_{\mathtt{r}} \end{cases}$$

$$(\mu X.G)\!\downarrow_{\mathtt{r}} = \begin{cases} \mu X.G\!\downarrow_{\mathtt{r}} & \text{if } \mathtt{r} \text{ occurs in } G \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$X\!\downarrow_{\mathtt{r}} = X \qquad \mathtt{end}\!\downarrow_{\mathtt{r}} = \mathsf{end}$$

Fig. 3. Projection of a global type G onto a participant $\mathtt{r}$ from PPDP'17.

4.4 A Reversible Semantics

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

Reversibility is useful in a concurrent setting when an error is encountered but the current state is invalid. In such a case the program should revert to its initial state, from which next steps can be considered (e.g. to retry or to log the failure).

The naive way to achieve reversibility in the semantics is to store a snapshot of the initial state, but the memory consumption of this method is too large for this method to be practical. 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.

The PPDP'17 paper provides a convenient approach tailored to the process calculus and local types that we have seen. For every forward step, just enough information is stored to undo it. Conceptually, we are leaving behind a trail of breadcrumbs so we can always find our way back.

The challenge, then, is to find what the minimal amount of information actually is for every instruction. The PPDP'17 paper tells us that 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 of them it stores a bit of extra information like the `owner`.

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

1. Used variable names in receives
   The rest of the program depends on the name that is assigned to a received value, like `decision` in the example below.
   When reverting, we need to reinstate the name that the rest of the program expects.

   ```
   decision <- H.receive
   H.send decision
   ```

   When we advance the snippet above, `decision` is internally renamed to `var1` and this binding is added to the store.

   ```
   H.send var1
   ```

   Now we want to move back. We can't simply create a new name: if we then move forward again, `var1` will not be defined.

   ```
   var2 <- H.receive
   H.send var1
   ```

   Instead, the receive needs to store the name of the value its result is bound to, in this case `var1`:

   ```
   var1 <- H.receive
   H.send var1
   ```

   But we also store the original name (given by the programmer) and actually restore that. The original name can helpful in debugging programs, to know what value is actually used:

   ```
   decision <- H.receive
   H.send decision
   ```

2. Unused branches
   When a choice is made and then reverted, we want all our options to be available again. Currently, another label cannot actually be selected after reverting: the selected label depends only on the values of variables in the program. Future work may make it possible to also use the failure information to influence the choice made.

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

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

The code above shows how the choices are stored. We need to remember which choice was made, and the order of the options is important. We use a `Zipper` to store the elements in order and use the central `a` to store the choice that was made.

3. 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 `Application` constructor so when stepping backward the function and the argument can be recovered.
   You might think that a stack would be a simpler solution, but a stack can give invalid behavior. Say that a participant is running in two locations, and the last-performed action at both locations is a function application. Now we want to undo both applications, but the order in which to undo them is undefined: we need both orders to work. When the application keeps track of exactly which function and argument it used the end result is always the same. Only using a stack could mix up the applications.
4. 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. Fig. 4 below illustrates this process:

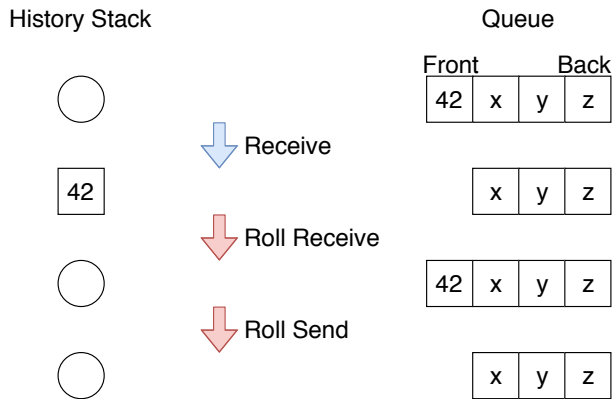

Fig. 4. Reversal of send and receive

1. **receive**: the value 42 is popped from the queue but pushed onto the history stack.

2. **roll receive**: Now when the receive is rolled, the value is moved back from the history stack onto the queue.
3. **roll send**: When the send is rolled the value is moved from the head of the queue into the sender's process.

4.5 Abstraction Passing is Protocol Delegation

Protocol delegation is where a participant can delegate (a part of) their protocol to be fulfilled by another participant. An example of where this idea is useful is a load balancing server: from the client's perspective, the server handles the request, but actually the load balancer delegates incoming requests to workers. The client does not need to be aware of this implementation detail.

Recall the definition of `ProgramF`:

```
data ProgramF value next
    -- communication primitives
    = Send
        { owner :: Participant
        , value :: value
        , continuation :: next
        }
    | ...
```

The `ProgramF` constructors that move the local type forward (send/receive, select/offer) have an `owner` field that stores whose local type they should be checked agains and modfiy. This field is also present in the `TypeContext`.

Because we allow references to values in the implementation, all references in a function have to be dereferenced before it can be safely sent over a channel.

5 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:

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

These functions take a `Location`, our way of modeling different threads or machines, and tries to move the process at that location forward or backward. The `Session` type contains the `ExecutionState`, the state of the session (all programs, local types, variable bindings, etc.), and can throw errors of type `Error`, for instance when an unbound variable is used.

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

See also Appendices A.6 on `StateT` and A.5 on `Except`

The semantics in the PPDP'17 paper guides how we store the execution state. Some data is bound to its location (for instance the process that is running) and other data is bound to its participant (for instance the local type).

The information about a participant grouped in a type called `Monitor`:

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

data Binding =
    Binding
        { _visibleName :: Identifier
        , _internalName :: Identifier
        }
    deriving (Show, Eq)
```

- `_localType` contains `TypeContext` and `LocalType` stored as a tuple. This tuple 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.
- `_usedVariables` and `_applicationHistory` are used in reversal. As mentioned in Section 4.4, used variable names need to be stored in order to be able to use them when reversing. We store them in a stack keeping both the original name given by the programmer and the generated unique internal name. For function applications we use a `Map` indexd by unique identifiers that stores function and argument.
- Finally `_store` is a variable store with the currently defined bindings. Variable shadowing - where two processes of the same participant define the same variable name - is not an issue, because variables are assigned a guaranteed unique name.

We can now 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 inspecting values, to see whether they are functions and if so, to extract their bodies for application.

5.1 Running and Debugging Programs

Finally, we want to be able to run our programs. The implementation offers mechanisms to step through a program interactively, and run it to completion.

We can step through the program interactively in the Haskell REPL environment (Appendix A.1). Assuming the `ThreeBuyer` example is loaded, we can print the initial state of our program:

```
> initialProgram
locations: fromList [("l1",("A",[],Fix (Send {owner = "A", ...
```

Next we introduce the `stepForward` and `stepBackward` functions. They use mutability, normally frowned upon in Haskell, to avoid having to manually keep track of the updated program state like in the snippet below:

```
state1 = stepForwardInconvenient "l1" state0
state2 = stepForwardInconvenient "l1" state1
state3 = stepForwardInconvenient "l1" state2
```

Manual state passing is error-prone and inconvenient. We provide helpers (that internally use `IORef`) to work around this issue. We must first initialize the program state:

```
> import Interpreter
> state <- initializeProgram initialProgram
```

Then we can use `stepForward` and `stepBackward` to evaluate the program:

```
> stepForward "l1" state
locations: fromList [("l1",("A",[],Fix (Receive {owner = "A", ...
> stepForward "l4" state
locations: fromList [("l1",("A",[],Fix (Receive {owner = "A", ...
```

When the user tries an invalid step, an error is displayed. for instance after l_1 and l_2 have been moved forward once like in the snippet above, l_1 cannot move forward (it needs to receive but there is nothing in the queue) and not backward (l_4, the receiver, must undo its action first).

```
> stepForward "l1" state
*** Exception: QueueError "Receive" EmptyQueue
CallStack (from HasCallStack):
  error, called at ...
> stepBackward "l1" state
*** Exception: QueueError "BackwardSend" EmptyQueue state
CallStack (from HasCallStack):
  error, called at ...
```

Errors are defined as:

```
data Error
    = UndefinedParticipant Participant
    | UndefinedVariable Participant Identifier
    | SynchronizationError String
    | LabelError String
    | QueueError String Queue.QueueError
    | ChoiceError ChoiceError
    | Terminated
```

To fully evaluate a program, we use a round-robin scheduler that calls `forward` on the locations in order. A forward step can produce an error. There are two error cases that we can recover from:

- **blocked on receive**, either `QueueError _ InvalidQueueItem` or `QueueError _ EmptyQueue`: the process wants to perform a receive, but the expected item is not at the top of the queue yet. In this case we want to proceed evaluating the other locations so they can send the value that the erroring location expects. The `_` in the patterns above means that we ignore the `String` field that is used to provide better error messages. Because no error message is generated, that field is not needed.
- **location terminates** with `Terminated`: the execution has reached a `NoOp`. In this case we do not want to schedule this location any more.

Otherwise we continue until there are no active (non-terminated) locations left. For code see Appendix A.11.

Running until completion (or error) is also available in the REPL:

```
> untilError initialProgram
Right locations: fromList [("l1",("A",[],Fix NoOp)), ...
```

Note that this scheduler can still get into deadlocks, for instance consider these two equivalent global types:

```
globalType1 = do
    GlobalType.transaction A V Title
    GlobalType.transaction V B Price
    GlobalType.transaction V A Price
    GlobalType.transaction A B Share

globalType2 = do
    GlobalType.transaction A V Title
    GlobalType.transaction V A Price
    GlobalType.transaction V B Price
    GlobalType.transaction A B Share
```

The second and third line are swapped. The communication they describe is the same, but in practice they are very different. The first one will run to completion, the second one can deadlock because `A` can send a `Share` before `V` does. `B` expects the share from `V` first, but the share from `A` is the first in the queue. Therefore, no progress can be made.

5.2 Properties of Reversibility

The main property that a reversable semantics 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.

The global type defines a partial order on all the communication steps. The relation of this partial order is causal dependency. Stepping backward is only allowed when all its causally dependent actions are undone. This guarantee is baked into every part of the semantics. For instance, a send can only be undone when the receive is already undone, because there is a data dependency between the two actions (the sent value).

Causal consistency is formally proven for the PPDP'17 semantics. We claim that the Haskell implementation is faithfull enough that causal consistency is preserved.

6 Convenient Syntax with the Free Monad

The types we have 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 Appendix A.7).

While a `Fix`ed data type is easy to manipulate and traverse, it can be messy to write. The program below implements "receive and bind the value to `result`, then send 42":

```
program =
    Fix
```

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

The syntax distracts from the goal of the program. `Program` and `GlobalType` are types that we will write a lot manually, so fixing this issue is important.

Conveniently, Haskell has a long tradition of embedded domain-specific languages. In particular we can use a cousin of `Fix`, the `Free` monad (Appendix A.9) to get access to do-notation (Appendix A.3). Concretely, the do-notation makes it possible instead of the above write:

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

Behind the scenes, the do-notation produces a value of type `HighLevelProgram a` using some helpers like `send` and `terminate`.

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

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

terminate :: HighLevelProgram a
terminate = liftF NoOp

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

The above snippet introduces some new Haskell concepts that require some background. Pure languages can simulate mutable state using a type called `State` (Appendix A.6). `State` is an instance of monad (Appendix A.4) which makes it easy to chain stateful computations. In this case our piece of state is a pair `(Participant, Int)`: the participant is the owner of the block, and the `Int` is a counter used to generate unique variable names. To combine `State` with `Free` (and to combine two monads in general) we need the `StateT` monad transformer (Appendix A.6).

In the `send` helper we use the unit type `()` as a placeholder or hole. A continuation will need to fill the hole eventually but it is not available yet. When a `HighLevelProgram` is converted into a `Program`, we want to be sure there are no remaining holes. In this particular case that means all branches must end in `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 Appendix A.10.

7 Discussion

7.1 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 primary invaluable feature is algebraic data types (ADTs) also known as tagged unions or sum types.

Compare 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
```

The definitions 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).

Secondly, purity and immutability are very valuable in implementing and testing the reversible semantics. The type system can actually guarantee that we have 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 (or to test for the domain you're interested in) that

```
f . g = identity && g . f = identity
```

In an impure language, even if the above equalities are observed we cannot be sure that there were no side-effects. Because we do not need to consider a context (the outside world) here, checking that reversibility works is as simple as comparing initial and final states for all backward transition rules.

7.2 Concurrent Debuggers

As mentioned we initially also set out to investigate how useful PPDP semantics are for debugging concurrent programs. As it stands, there are two missing features

1. Modifying concurrent control flow
   That is, a way to specify which thread will (try to) take a step forward next. The problem with concurrency is not so much technical: the primitives are available. What is needed is some way to step through a program one instruction at a time. The real challenge is providing a convenient mechanism for doing so.
2. Convenient user interface
   CaReDeb provides a command line interface. While CaReDeb is interesting from a technical point of view, its interface is not convenient. Additionally, the overhead of bringing the problem into CaReDeb's language is large: more time is probably spent translating than actually debugging.
   We think that a good graphical interface is possible, but besides technical features a good user experience also needs user feedback. This means that we need to look for users and concrete use cases. It would help if there were some concrete set of problems that could be extracted from code, compiled into a format/language that our debugger understands and then visually debugged. Performance can then be evaluated based on how well the debugger solves real-world problems.

8 Conclusion

We have given an encoding of the PPDP'17 semantics in the Haskell programming language. By embedding the semantics we can now run and verify our example programs automatically and inspect them interactively.

We have seen that the formal programming model in the PPDP'17 paper can be split into three core components: a process calculus, multiparty session types and reversibility. Additonally, all three of these are representable as recursive Haskell data types. All other features of PPDP'17 (the authors note decoupled rollbacks and abstraction passing, including delegation) can easily be integrated when these three types are established.

Relatedly, the implementation proces has shown that sticking to the formal implementation leads to better code. There is less space for bugs to creep in.

Furthermore, Haskell's mathematical nature means that the implementation inspired by the formal specification is easy (and often idiomatic) to express.

We have seen that Haskell allows for the definition of flexible embedded domain-specific languages, and makes it easy to transform between different representations of our programs using among others `Monad.Free`.

Finally, we have discussed how this implementation can be used for concurrent debugging.
With more focus on the user experience, a solid concurrent debugging can be built on the foundations presented here.

The work in this thesis has been presented at the TFP'18 conference in Gothenburg, Sweden. Attending the conference was made possible by financial support from the undergratudate school of science and the Bernoulli Institute, for which we are extremely grateful.

References

1. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

2. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

3. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *POPL 2008*, pages 273–284. ACM, 2008.

4. I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014.

5. C. A. Mezzina and J. A. Pérez. Causally consistent reversible choreographies: a monitors-as-memories approach. In W. Vanhoof and B. Pientka, editors, *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, pages 127–138. ACM, 2017.

6. F. van Walree. Session types in cloud haskell. Master's thesis, 2017.

A Appendix

Background information on Haskell syntax and concepts used in the thesis.

A.1 Installing and Running

This project is written in Haskell and built with its *stack* build tool. Stack can be downloaded from here. The next step is to clone the repository, which is available on github.

The snippet below will clone the repository (assumes ssh is set up) and build it. This can take a while because it also has to download and install the Haskell GHC compiler.

```
git clone git@github.com:folkertdev/reversible-debugger.git
cd reversible-debugger
stack build
```

Finally we can load an example program in the REPL:

```
stack ghci # opens the interactive environment
:l src/Examples/ThreeBuyer.hs # loads the example
```

A.2 Performing IO in Haskell

One of Haskell's key characteristics is that it is pure. This means that our computations cannot 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 computers 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 cannot 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 have 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.

A.3 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.

A.4 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 have 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 will look at some of the types used in the code for this thesis.

A.5 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.

A.6 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.

```haskell
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 do not naturally combine: `m1 (m2 a)` may not have a law-abiding monad instance.

```haskell
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.*

A.7 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.

A.8 Fix

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

```
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

```
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

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

Defining values of type `Fix ExprF` requires wrapping of constructors with `Fix`. We will describe a better way of writing `Fix`ed values in Section 6

```
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

```
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.

A.9 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)
```

A.10 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 does not introduce extra language extensions to the project.

A.11 Scheduling code

```
data Progress = Progress | NoProgress deriving (Eq, Show)

round :: List Location -> ExecutionState Value
      -> Either Error ( List (Location, Progress)
                      , ExecutionState Value
                      )
round locations state =
    foldM helper [] locations
        |> flip State.runStateT state
        |> Except.runExcept
  where helper :: List (Location, Progress) -> Location
```

```
            -> Session Value (List (Location, Progress))
        helper accum location = do
            state <- State.get

            let evaluated =
                    State.runStateT (forward location) state
                        |> Except.runExcept

            case evaluated of
                Right ( _, s ) -> do
                    State.put s
                    return $ ( location, Progress ) : accum

                Left (QueueError origin (InvalidQueueItem message)) ->
                    -- blocked on receive, try moving others forward
                    return $ ( location, NoProgress ) : accum

                Left (QueueError origin EmptyQueue) ->
                    -- blocked on receive, try moving others forward
                    return $ ( location, NoProgress ) : accum

                Left Terminated ->
                    return accum

                Left err ->
                    -- other errors are raised
                    Except.throwError err


untilError :: ExecutionState Value
           -> Either Error (ExecutionState Value)
untilError state@ExecutionState{ locations } =
    helper (Map.keys locations) state
  where helper locations state = do
            ( locationProgress, newState ) <-
                Interpreter.round locations state

            let isProgress =
                    any
                        (\(_, progress) -> progress == Progress)
                        locationProgress

            if null locationProgress then
                -- no active locations
                Right state
```

```
else if isProgress then
    helper (List.map fst locationProgress) newState

else
    error $ "DEADLOCK\n" ++ show state
```