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.

0.1 Global Type

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

```haskell
type GlobalType u = Fix (GlobalTypeF u)

data GlobalTypeF u next
    = Transaction
        { from :: Participant
        , to :: Participant
        , tipe :: u
        , continuation ::  next
        }
    | Choice
        { from :: Participant
        , to :: Participant
        , options :: Map String next
```

[*] F. de Vries is a BSc student.

```
        }
    | R next
    | V
    | Wk next
    | End
    deriving (Show, Functor)
```

0.2 Local Type

$$U, U' ::= \texttt{bool} \mid \texttt{nat} \mid \cdots \mid T \to \diamond$$

$$T, T' ::= \mathrm{p}!\langle U \rangle.T \mid \mathrm{p}?\langle U \rangle.T \mid \mathrm{p}\oplus\{l_i : T_i\}_{i \in I} \mid \mathrm{p}\&\{l_i : T_i\}_{i \in I} \mid \mu X.T \mid X \mid \mathsf{end}$$

```
data LocalTypeF u f
    = Transaction (Transaction u f)
    | Choice (Choice u f)
    | Recursion (Recursion f)
    | End

data Transaction u f
    = TSend
        { owner :: Participant , receiver :: Participant
        , tipe :: u
        , continuation :: f
        }
    | TReceive
        { owner :: Participant , sender :: Participant
        , tipe ::  u
        , continuation :: f
        }

data Choice u f
    = COffer
        { owner :: Participant
        , selector :: Participant
        , options :: Map String (LocalType u)
        }
    | CSelect
        { owner :: Participant
        , offerer :: Participant
        , options :: Map String (LocalType u)
        }
```

```
data Recursion f = R f | V | Wk f
```

0.3 Values

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

The implementation's values are much more elaborate. The main addition is operators, that can be used for arithmetic and comparison. Comparison is needed to let runtime values influence choices (`select` and `if-then-else`).

```
data Value
    = VBool Bool
    | VInt Int
    | VString String
    | VIntOperator Value IntOperator Value
    | VComparison Value Ordering Value
    | VUnit
    | VFunction Identifier (Program Value)
    | VReference Identifier
    | VLabel String
```

0.4 Process/Program

The formal definition is given by

$$
\begin{array}{lll}
P, Q & ::= & u!\langle V\rangle.P \;\big|\; u?(x).P \;\big|\; u \triangleleft \{l_i.P_i\}_{i\in I} \;\big|\; u \triangleright \{l_i : P_i\}_{i\in I} \\
 & \big| & P \mid Q \;\big|\; X \;\big|\; \mu X.P \;\big|\; V\,u \;\big|\; (\nu\, n)P \;\big|\; \mathbf{0}
\end{array}
$$

Question: Function application is `Value -> Name -> Process`. Is there a reason the argument is only a name, and not a `Value`?

Which is encoded as this data type

```
data ProgramF value next
    -- passing messages
    = Send { owner :: Participant, value :: value, continuation :: next }
    | Receive { owner :: Participant, variableName :: Identifier, continuation :: next  }

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

    | Parallel next next
    | Application Participant Identifier value
    | NoOp

    -- recursion omitted, see note

    -- syntactic sugar for better examples
    | Let Participant Identifier value next
    | IfThenElse Participant value next next
```

The four communication operations, `send`, `receive`, `offer` and `select` are constructors. Likewise, parallel execution, function application and unit (the empty program) have their own constructors.

To create more interesting examples, we've also introduced constructors for let-bindings and if-then-else statements. These constructions can be reduced to function applications, so they don't add new semantics. They are purely syntactic sugar, but do show how one might extend this language with new constructors.

Name restriction is not needed because we use generated variable names that are guaranteed to be unique.

because values already allow recursion, process recursion can be implemented with value recursion and function calls, to the point that we've defined

```
recursive :: (HighLevelProgram a -> HighLevelProgram a) -> HighLevelProgram a
recursive body = do
    thunk <- recursiveFunction $ \self _ ->
        body (applyFunction self VUnit)

    applyFunction thunk VUnit
```

0.5 Type Context

Now that we've defined session types and programs that can go forward, we need to construct the memory that allows us to go backward. The first step is backward types.

Definition 1. *Let $k, k', \ldots$ denote fresh name identifiers. We define type contexts as (local) types with one hole, denoted "$\bullet$":*

$$\begin{aligned}\mathbb{T}, \mathbb{S} \quad ::= \quad & \bullet \;\Big|\; q \oplus \{l_w : \mathbb{T}\,;\; l_i : S_i\}_{i \in I \setminus w} \;\Big|\; q \& \{l_w : \mathbb{T}\,,\; l_i : S_i\}_{i \in I \setminus w} \\ & \Big|\; \alpha.\mathbb{T} \;\Big|\; k.\mathbb{T} \;\Big|\; (\ell, \ell_1, \ell_2).\mathbb{T}\end{aligned}$$

The hole is where the remaining (i.e. future) local type goes. Keeping the two structures in a tuple, we can move forward and backward though the local type (as long as there are actions to perform).

In the formal definition, the most outer tag is the least-recent action, and the tag closest to the hole is the most-recent action. In practice, we only want to look at the most-recent action: it is the first action we need to reverse. Because pattern matching is more efficient at the top, we invert the type so the hole is implicit and the most-recent element is at the top.

```
data TypeContextF a previous
    = Empty

    | Transaction (LocalType.Transaction a previous)
    | Selected
        { owner :: Participant
        , offerer :: Participant
        , selection :: Zipper (String, LocalType a)
            , continuation :: previous
        }
    | Offered
        { owner :: Participant
        , selector :: Participant
        , picked :: Zipper (String, LocalType a)
            , continuation :: previous
        }

    | Application Participant Identifier previous
    | Spawning Location Location Location previous

    | R previous
    | Wk previous
    | V previous

    -- sugar
    | Branched { owner :: Participant, continuation :: previous }
    | Assignment { owner :: Participant, continuation :: previous }
    deriving (Eq, Show, Generic, Functor, Foldable, Traversable)
```

The `Zipper` data type stores the labels and types for each option in-order. A zipper is essentially a triplet `(List a, a, List a)`. The order is important in the implementation because every option comes with a condition, and the first option that evaluates to `True` is picked.

The `Branched` and `Assignment` constructors are empty tags - because those operations don't influence the type.

The `TypeContext` doesn't store any program/value information, only type information

We use pattern synonyms to give better names to the options and deal with the `Fix` wrapper.

0.6 Reversing programs

The different instructions need different information to be stored in order to be reversible. There's also a number of different places where it makes sense to store that data. We use four concepts:

Free Variable Stack

Bit of a misnomer, because it's actualy a stack of used variable names. When reversing a `receive` or `let`-binding, we use it to get the name the programer originally used and the name that internally was assigned to the value.

Program Stack

A stack used to store pieces of program that aren't evaluated: The remaining options in a `select` or `offer`, or the other case in an `ifThenElse`.

Application History

A map that stores the function and the argument of a function application.

History Queue

Whenever an element is `received`, the element isn't acually removed from the message queue and voided, but moved to the history queue. When reversing, we use the history queue to verify the sender, receiver and type.

0.7 Monitor

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

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

The monitor uses a type context, where the hole is substituted with a local type. To store the structure with the hole, and the value that shoudl go in the hole, we use a 2-tuple to combine them.

The PPDP paper omits reduction rules for recursive local types, but in the actual implementation we need to keep track of the recursion points, and what the type from that point would be, so we can later return to them. The application history is also moved into the monitor.

The free (i.e. used) variable list is stored in the monitor. Our variables are globally unique, so technically we could store this list globally, but we've chosen to follow the PPDP implementation here.

0.8 Global State

Finally we need some global state that contains all the programs and types and the queue and such.

```
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)
        }

data OtherOptions
```

```
    = OtherSelections (Zipper (String, Value, Program Value))
    | OtherOffers (Zipper (String, Program Value))
    | OtherBranch Value Bool (Program Value)
    deriving (Show, Eq)
```

0.9

type RunningFunction = (Value, Value, Location)

type RunningFunctions = Map K RunningFunction

data Message = Value Value | Label String

data Tag = Empty | Full

type QueueHalf = List (Participant, Participant, Message)

data Alpha u = Send u | Receive u

data LocalType u = End | SendReceive (Alpha u) (T u) | Select Label (T u) | Offer Label (T u)

data LocalHistoryType u = Before T | After T | AfterSendsAndReceives (List Alpha) (T u) | Select (Label, (T u)) (Map Label (LocalHistoryType u)) | Offer (Label, (T u)) (Map Label (LocalHistoryType u))

– we've removed recursion from the process calculus, so we can drop the free variable list

data Monitor u = Full (TypeContext u, LocalType u, Map Identifier Value) | Empty (TypeContext u, LocalType u, Map Identifier Value)

type K = Identifier

data TypeContext u = Hole | Offer (Label, TypeContext u) (Map Label (T u) | Select (Label, TypeContext u) (Map Label (T u) | SendOrReceive (Alpha u) | Application K (TypeContext u) | Spawned (Location, Location, Location) (TypeContext u)

1 Scratchpad

1.1 Intro

My work is a practical implemenation of the paper “reversible session-based concurrency”

Core topics:

- the pi-calculus: a calculus for concurrent computation
- session types: a type system for concurrent computation
- reversibility

1.2 the pi-calculus

The pi-calculus describes concurrent computation, much like the lambda calculus describes computation.

The lambda calculus at its core has two concepts

function creation: `\x -> x` function application `(\x -> x) 42 => 42`

The pi-calculus defines

- send: `x<y>.P` send value y over channel x, then run P
- receive: `x(y).P` receive on channel x, bind the result to y, then run P
- parallel: `P|Q` run P and Q simultaneously

And an extra reduction rule

`x<a>.P | x(b).Q => P | Q`

Now because this is math and we can make up whatever rules we like (as long as they're consistent), we can also introduce choice

`P + Q`

We'll revisit choice later

questions

- how do pi and lambda relate: wiki says that lambda is encodable in pi, is there some intuition for that?
- how should choice be introduced here
- what makes the higher-order pi-calculus special (guess: sending of processes over channels). Does it need further explanation

1.3 Session types

The types we're used to (Int, String, Maybe) prevent us from doing stupid things with our data. Session types prevent us from doing stupid communication. They allow us to embed protocols into our programs.

A session type can express something like

"I will send Bob a bool, and then I expect and int from Alice"

In the 2-party case, the types are exactly dual, but in a multi-party scenario type becomes more complex

A choice at the process level can also mean a choice between two (or more) session types. In this case, the choice needs to be communicated to everyone (because one of the branches might recurse and then everyone has to recurse).

1.4 Static vs. Dynamic

If this were a perfect world, we'd of course all use static languages with a strong, expressive type system. Alas, that's not our timeline. In the `RealWorld` there are systems written in different languages, and to check their communication we need dynamic, runtime session types.

(if someone can tell me more about static session types and whether the linear types in haskell can let us write them without indexed monads, please come talk to me after)

In any case, we'll look at dynamic session types from now on

1.5 Reversibility

reversibllity means that we can move backward in the execution of a program. This is convenient when some error occurs at runtime, and we want to roll (undo) a full transaction.

We achieve reversibility by construction by ensuring that every forward step has a corresponding backward step. We store all needed details for backward steps. Storing the full previous program states takes too much memory (in theory)

This is where haskell as an implementation language really shines: because of immutability, it is very easy to undo all modifications and then test that `backward . forward = identity`.

1.6 our session types

The protocol is defined as a global type

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

We can use haskell union types to give some extra safety, typos in the member or type names are compiler errors.

Which is then projected onto the participants of the protocol The projection contains only the actions that are relevant for the participant. in pseudo-code (this is all generated under the hood).

```
localType :: LocalType.LocalType MyParticipants MyType
localType = LocalType.localType $ do
    LocalType.sendTo V Title
    LocalType.receiveFrom V Price
    LocalType.sendTo B Share
    LocalType.receiveFrom B Ok
    LocalType.end
```

The full datatype for `GlobalType` is ... Note the three constructors for recursion. R defines a recursion point, a point that we can jump back to. V is the recursion variable, and takes us back to a recursion point. Wk weakens the recursion, and allows us to jump to less tightly-bound Rs.

1.7 Our process calculus

there is of course send and receive. There is also offer and select, for making choices at runtime and communicating one's choice to other participants. Then we have function application and a NoOp, and let-bindings and ifThenElse. These last two are encodable as functions and thus purely syntactic sugar.

Recursion is possible with functions:

We decouple sending and receiving by using a message queue that buffers the messages

1.8 A DSL with the free monad

Writing examples with just the data type is very tedious. We can use a free monad DSL to make the process a bit more pleasant.

```
alice = do
    let share = VInt 42
    H.send (VString "accursedUnutterablePerformIO" )
    price <- H.receive
    H.send share
    ok <- H.receive
    H.terminate
```

1.9 reversibility in types

for the types we have a dual `TypeContext` that stores all the type information we need to revert. A tuple (TypeContext, LocalType) thus stores exactly where we are in the type, and we can move forward and backward with it.

1.10 reversibility in programs

We need to store a bunch of stuff

- used variable names
- unused branches in `select`, `offer` and `if-then-else`
- function applications: the function and the argument
- messages

1.11 the Monitor and Synchronization

the monitor stores the type and variables for a participant

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

When we roll a coordinated action (send/receive, select/offer), we have to make sure that both participants are synchronized: the action we want to roll is their most-recent action.

Once processes are synchronized, they must first actually roll the action before they can do anything else. The advantage of this approach is that rolling is decoupled.

1.12 ExecutionState

Finally we wrap everything into an `ExecutionState` that keeps track of all the types, programs, the queue and some counters used to generate unique identifiers.

```
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)
        }
```

1.13 Conclusion

We've seen . . .