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.

1 Introduction

We implement the model in 1.

2 The Process Model

I think we need to explicitely define

- location
- participant
- queue

3 Our Haskell Implementation

We set out to implement the language, types and semantics given above. The end goal is to implement the two stepping functions

* F. de Vries is a BSc student.

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

Where `Session` contains an `ExecutionState` holding among other things a store of variables, and can fail producing an `Error`.

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

Additionally we need to provide a program for every participant, a monitor for every participant and a global message queue. All three of those need to be able to move forward and backward.

TODO list explictly the next sections and what they describe

3.1 Global and Local Types

As mentioned, we have two kinds of session types: Global and Local. The Global type describes interactions between participants, specifically the sending and receiving of a value (a transaction), and selecting one out of a set of options (a choice). The definition of global types is given by

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

The recursive constructors are taken from 2. `R` introduces a recursion point, `V` jumps back to a recursion point and `Wk` weakens the recursion, making it possible to jump to a less tightly-binding `R`.

A global type can be projected onto a participant, resulting in that participant's local type. The local type describes interactions between a participant and the central message queue. Specifically, sends and receives, and offers and selects. The projection of `globalType` onto `A` is equivalent to this pseudo-code of `derivedTypeForA`.

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

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

derivedTypeForA :: LocalType MyType
derivedTypeForA = do
    send V Title
    receive V Price
    send B Share
    receive B Ok
```

3.2 A Language

We need a language to use with our types. It needs at least instructions for the four participant-queue interactions, a way to assign variables, and a way to define and apply functions.

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

type Program value = Fix (ProgramF value)

data ProgramF value next
    -- transaction 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
    | Let Identifier value next
    | IfThenElse value next next
    | Literal value
    | NoOp
    deriving (Eq, Show, Functor)


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
    deriving (Eq, Show)
```

In the definition of `ProgramF`, the recursion is factored out and replaced by a type parameter. The `Fix` type reintroduces the ability to create to create arbitrarily deep trees of instructions. The advantage of using `Fix` is that we can use recursion schemes - like folds - on the structure.

Given a `LocalType` and a `Program`, we can now step forward through the program. For each instruction, we check the session type to see whether the instruction is allowed.

3.3 An eDSL with the free monad

We create an embedded domain-specific language (eDSL) using the free monad to write our example programs. This method allows us to use haskell's do-notation, which is much more convenient than writing programs with `Fix`.

The free monad is a monad that comes for free given some functor. The `ProgramF` type is specifically created in such a way that it is a functor in `next`. The idea then is to use the free monad on our `ProgramF` data type to be able to build a nice DSL.

For the transformation from `Free (ProgramF value) a` back to `Fix (ProgramF value)` we need also need some state: a variable counter that allows us to produce new unique variable names.

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

uniqueVariableName :: HighLevelProgram Identifier
uniqueVariableName = do
    (location, participant, n) <- State.get
    State.put (location, participant, n + 1)
    return $ "var" ++ show n

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

receive :: HighLevelProgram Value
receive = do
    (_, participant, _) <- State.get
    variableName <- uniqueVariableName
    HighLevelProgram $ lift $ liftFree (Receive participant variableName ())
    return (VReference variableName)

terminate :: HighLevelProgram a
terminate = HighLevelProgram (lift $ Free NoOp)
```

We can now write a correct implementation of `As` local type.

```haskell
aType :: LocalType MyType
aType = do
    send V Title
    receive V Price
    send B Share
    receive B Ok

alice = H.compile "Location1" "A" $ do
    let share = VInt 42
    H.send (VString "address" )
    price <- H.receive
    H.send share
    ok <- H.receive
    H.terminate
```

`HighLevelProgram` is transformed into a `Program` by evaluating the `StateT`, this puts the correct owner and unique variable names into the tree, and then transforming `Free` to `Fix` by putting `NoOp` at the leaves where needed.

```
freeToFix :: Free (ProgramF value) a -> Program value
freeToFix (Pure n) = Fix NoOp
freeToFix (Free x) = Fix (fmap freeToFix x)

compile :: Location -> Participant -> HighLevelProgram a -> Program value
compile location participant (HighLevelProgram program) =
    freeToFix $ runStateT program (location, participant, 0)
```

3.4 Ownership

The `owner` field for send, receive, offer and select is makes sure that instructions in closures are attributed to the correct participant.

```
bob = H.compile "Location1" "B" $ do
    thunk <-
        H.function $ \_ -> do
            H.send (VString "Lucca, 55100")
            d <- H.receive
            H.terminate

    price <- H.receive
    share <- H.receive
    let verdict = price `H.lessThan` VInt 79
    H.send verdict
    H.send verdict
    H.send share
    H.send thunk


carol = H.compile "Location1" "C" $ do
    h <- H.receive
    code <- H.receive
    H.applyFunction code VUnit
```

Here `B` creates a function that performs a send and receive. Because the function is created by `B`, the owner of these statements is `B`, even when the function is sent to and eventually evaluated by `C`.

The design of the language and semantics poses some further issues. With the current mechanism of storing applications, functions have to be named. Hence `H.function` cannot produce a simple value, because it needs to assign to a variable and thereby update the state.

It essential that all references in the function body are dereferenced before sending. References that remain are invalid at the receiver, causing undefined behavior.

3.5 Reversibility

Every forward step needs an inverse. The inverse needs to contain all the information needed to recreate the instruction and local type that performed the forward step.

```
type TypeContext program value a = Fix (TypeContextF program value a)

data TypeContextF program value a f
    = Hole
    | LocalType (LocalTypeF a ()) f
    | Selected
        { owner :: Participant
        , offerer :: Participant
        , selection :: Zipper (String, value, program, LocalType a)
        , continuation :: f
        }
    | Offered
        { owner :: Participant
        , selector :: Participant
        , picked :: Zipper (String, program, LocalType a)
        , continuation :: f
        }
    | Branched
        { condition :: value
        , verdict :: Bool
        , otherBranch :: program
        , continuation :: f
        }
    | Application Identifier Identifier f
    | Assignment
        { visibleName :: Identifier
        , internalName :: Identifier
        , continuation :: f
        }
    | Literal a f
    deriving (Eq, Show, Generic, Functor)
```

For **send/receive** and **offer/select**, the instructions that modify the queue, we must also roll the queue. Additionally, both participants must be synchronized. Synchronization ensures that both parties roll their parts, but the rolling can still happen in a decoupled way. The synchronization is a dynamic check that throws an error message if either participant is not in the expected state.

Rolling a **let** removes the assigned variable from the store. While strictly necesarry to maintain causal consistency, it is good practice.

Function applications are treated exactly as in the formal semantics: A reference is stored to the function and its arguments, so we can recreate the application later.

Given a `LocalType` and a `Program` we can now step through the program while producing a trace through the execution. At any point, we can move back to a previous state.

3.6 Putting it all together

At the start we described the `Session` type.

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

We now have all the pieces we need to define the execution state. Based on the error conditions that arise in moving forward and backward, we can also define a meaningful `Error` type.

3.6.1 The Monitor A participant is defined by its monitor and its program. The monitor contains various metadata about the participant: variables, the current state of the type and some other information to be able to move backward.

In the formal semantics we've defined a monitor as a tagged triplet containing a history session type, a set of free variables and a store assigning these variables to values. In the haskell implementation, the tag is moved into the history type `LocalTypeState`. The set of free variables and the store is merged into a dictionary. The set of variables is the set of keys of the dictionary.

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

Additionally, the application history is folded into the monitor as this is a participant-specific piece of data. Similarly, the monitor keeps track of recursion points to restore the local type when a recursive step is reversed.

3.6.2 ExecutionState The execution state contains the monitors and the programs, but also

- A variable and application counter to generate unique new names
- The central message queue.
- The `_isFunction` field, a function that extracts the function body used for function application.

```
data Queue a = Queue
    { history :: List ( Participant, Participant, a)
    , current :: List (Participant, Participant, a)
    }
    deriving (Eq, Show)

data ExecutionState value =
    ExecutionState
        { _variableCount :: Int
        , _applicationCount :: Int
        , _participants :: Map Participant (Monitor value String)
        , _locations :: Map Location (Map Participant (Program value))
        , _queue :: Queue value
        , _isFunction :: value -> Maybe (Identifier, Program value)
        }
```

3.6.3 Error generation There are a lot of potential failure conditions in this system. A small error somewhere in either the global type or the program can quickly move program and type out of sync.

Having descriptive error messages that provide a lot of context makes it easier to fix these issues.

```
data Error
    = UndefinedParticipant Participant
    | UndefinedVariable Participant Identifier
    | SynchronizationError String
    | LabelError String
    | QueueError String Queue.QueueError
    deriving (Eq, Show)
```

Not all errors are fatal. For instance, it is possible that one participant expects to receive the value, but it's not been sent yet. This will give a queue error - because the correct value is not in the queue - but the program and the type are perfectly fine. Implementing a workflow for error recovery is left as future work.

3.6.4 Stepping functions

We can now implement the stepping functions

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

A sketch of the implementation for forward:

- Given a `Location` and a `Participant` we can query the `ExecutionState` to get a `Program` and a `Monitor`.
- We pattern match on the program and the local type. If then match then we get into the business logic, otherwise we throw an error.
- The business logic must construct a new program and a new monitor. In the below example, a variable assignment is evaluated. Assignments don't affect the local type so the type is matchted against the wildcard `_`.

Next the assignment is added to the history type, then the monitor is updated with this new local type and an updated store with the new variable. The remaining program is updated by renaming the variable.

```
(Let visibleName value continuation, _) -> do
    variableName <- uniqueVariableName

    let newLocalType =
            LocalType.createState (LocalType.assignment visibleName variableName previous) :

        renamedValue = renameValue visibleName variableName value
        newMonitor =
            monitor
                { _store = Map.insert variableName renamedValue (_store monitor)
                , _localType = newLocalType
                }

    setParticipant location participant ( newMonitor, renameVariable visibleName variableNa
```

4 Concluding Remarks and Future Work

- also store the currenent position in the global protocol and use it to step
- UI/UX for working around errors
- make informed decisions when a branch of a choice fails
- integrating choice and recursion

Bibliography

[1] C. A. Mezzina and J. A. Pérez, "Causally consistent reversible choreographies: A monitors-as-memories approach," in *Proceedings of the 19th international symposium on principles and practice of declarative programming, namur, belgium, october 09 - 11, 2017*, 2017, pp. 127–138.

[2] F. van Walree, "Session types in cloud haskell," 2017.