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 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.

```
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)
```

Next we will look at how session types are represented, what the language looks like and how to keep track of and reverse past actions.

3.2 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)
```

```haskell
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`.

```haskell
a = "Alice"
b = "Bob"

data MyType
    = Address
    | ZipCode

globalType :: GlobalType MyType
globalType = do
    transaction a b ZipCode
    transaction b a Address
```

The Global type can then be projected onto a participant, resulting in a 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` and `b` is equivalent to:

```haskell
aType :: LocalType MyType
aType = do
    send b ZipCode
    receive b Address

bType :: LocalType MyType
bType = do
    receive a ZipCode
    send a Address
```

3.3 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.

```
type Participant = String
type Identifier = String

type Program = 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 to make interesting examples
    | 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. We then use `Fix` to give us back arbitrarily deep trees of instructions. The advantage of this transformation 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.

3.4 An eDSL with the free monad

Writing programs with `Fix` everywhere is tedious, and we can do better. We can create an embedded domain-specific language (eDSL) using the free monad. The free monad is a monad that comes for free given some functor. With this monad we can use do-notation, which is much more pleasant to write.

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.

```
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 give correct implementations to the local types given above.

```
aType = do
    send B ZipCode
    receive B Address

alice = do
    let zipcode = VString "4242AB"
    send zipcode
    address <- receive
    terminate

bType = do
    receive A ZipCode
    send A Address

bob = do
    zipcode <- receive
    let address = "mûnewei 42"
    send address
```

And then transform them into a `Program` with

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

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

3.5 Ownership

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

```
globalType = do
    transaction "B" "C" "thunk"
    transaction "B" "A" "address"
    transaction "A" "B" "amount"

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

```haskell
    H.send thunk

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

alice = H.compile "Location1" "A" $ do
    address <- H.receive
    H.send (VInt 42)
```

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.

3.6 Reversibility

Every forward step needs an inverse. When taking a forward step we store enough information to recreate the instruction and local type that made us perform the forward step.

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

data TypeContextF program value a f
    = Hole
    | SendOrReceive (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
    | Spawning Location Location Location f
    | Assignment
        { visibleName :: Identifier
        , internalName :: Identifier
        , continuation :: f
        }
    | Literal a f
    deriving (Eq, Show, Generic, Functor)
```

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

3.6.1 Synchronization

The formal semantics describe synchronizations before roling a transaction or a choice: both parties must be ready to roll the action. This condition is checked dynamically; having it as a transition rule doesn't really work.

3.7 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.

3.7.1 ExecutionState

The execution state contains the monitors and the programs at all locations. Additionally it countains a variable counter to generate unique new names, and the central message queue

```
type Queue a = [a]

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)
        }
```

3.7.2 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. Therefore, returning detailed error messages is required.

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

3.7.3 ?

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

4 Concluding Remarks and Future Work

- also store the currenent position in the global protocol and use it to step
- make informed decisions when a branch of a choice fails

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.