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

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

```
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

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

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

```
data LocalTypeF u f
    = Transaction (Transaction u f)
    | Choice (Choice u f)
    | Atom (Atom f)

data Transaction u f
    = TSend
        { owner :: Participant , receiver :: Participant
        , tipe :: u
        , continuation :: f
        }
    | TReceive
        { owner :: Participant , sender :: Participant
        , tipe ::  u
        , continuation :: f
        , names :: Maybe (Identifier, Identifier)
        }

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 Atom f = R f | V | Wk f | End
```

0.3 Local Type with History

This is where I'm less sure and it looks like the paper version is simpler than what I have.

0.4 Values

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

```
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
```

0.5 Process/Program

The formal definition is given by

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

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

```haskell
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 slightly 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

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

    applyFunction thunk VUnit
```

0.6 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.

Formal definition:

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

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

Data type

```
data TypeContextF a f
    = Hole
    | LocalType (LocalTypeF a ()) f
    | Selected
        { owner :: Participant
        , offerer :: Participant
        , selection :: Zipper (String, LocalType a)
        , continuation :: f
        }
    | Offered
        { owner :: Participant
        , selector :: Participant
        , picked :: Zipper (String, LocalType a)
        , continuation :: f
        }
    | Application Participant Identifier f
    | Spawning Location Location Location f

    -- sugar
    | Branched { continuation :: f }
    | Assignment { owner :: Participant, continuation :: f }
```

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

0.7 Reversing programs

There are a couple of ways that information is stored

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, but moved to the history queue.

0.8 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.

0.9 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)
```

0.10

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 Questions

- We discussed that function recursion is equivalent to Process recursion. With that, I think the free variable store can be dropped from the monitor. Is that correct
- Have we ever discussed configurations? What do they add
- Similarly, the evaluation and the general contexts. Are they useful in practice or only used for proofs?
- In section 2.2.2: "We require auxiliary definitions for **contexts**, stores, and type contexts." Which context is meant there (general or evaluation). Is this important?
- I think I've mixed up or merged "type contexts" and "local types with history". Not sure what's going on, but in the definition of monitors `H` is used, defined (in fig. 4) as "local types with history". But then in the semantics, the H is replaced by T[S] , with T being a "local type context" and S a normal local type (no history).
  Where did that `H` go? It never seems to be used in the semantics
- I can't find any machinery for recursive local types. When passing a μ, you need to remember that point (and the remaining type) to later return to it. Does this happen anywhere?
- With the PPDP semantics, can choice ever do something interesting? I feel like there needs to be a way to have values at runtime influence the choice made, but there is no mechanism for that.
- We should go over how function creation/calling works. I'd like to turn let-bindings and ifThenElse into function applications, but I'm not sure whether we can.

– guarded vs non-guarded choice.