

Reversible Session-Based Concurrency in Haskell*

Folkert de Vries¹ and Jorge A. Pérez¹[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

$$G, G' ::= p \rightarrow q : \langle U \rangle . G \mid p \rightarrow q : \{l_i : G_i\}_{i \in I} \mid \mu X . G \mid X \mid \text{end}$$

```
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

$$U, U' ::= \text{bool} \mid \text{nat} \mid \dots \mid T \rightarrow \diamond$$

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

```

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

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

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

```
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

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

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

```

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

  -- recursion omitted, see note

  | Application Participant Identifier value
  | NoOp

  -- syntactic sugar for better examples
  | Let Participant Identifier value next
  | IfThenElse Participant value next next
  | Literal value -- needed to define multi-parameter functions

```

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

```

0.6 Monitor

0.7

```

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.