

Reversible Session-Based Concurrency

An implementation in Haskell

Folkert de Vries (University of Groningen, NL)
Jorge A. Pérez (University of Groningen, NL)

University of Groningen, The Netherlands,

TFP 2018
Göteborg, Sweden
June 11, 2018

Concurrency is hard

Core Concepts

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

The pi-calculus

The lambda calculus at its core has two concepts

- ▶ function creation: $\lambda x \rightarrow x$
- ▶ function application: $f x$
- ▶ reduction: $(\lambda x \rightarrow x) 42 \Rightarrow 42$

The pi-calculus defines

- ▶ send: $\bar{x} < y >. P$ 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
- ▶ reduction: $\bar{x} < a >. P \mid x(b). Q \Rightarrow P \mid Q$

Session Types

Data types prevent us from making silly mistakes with **data**.

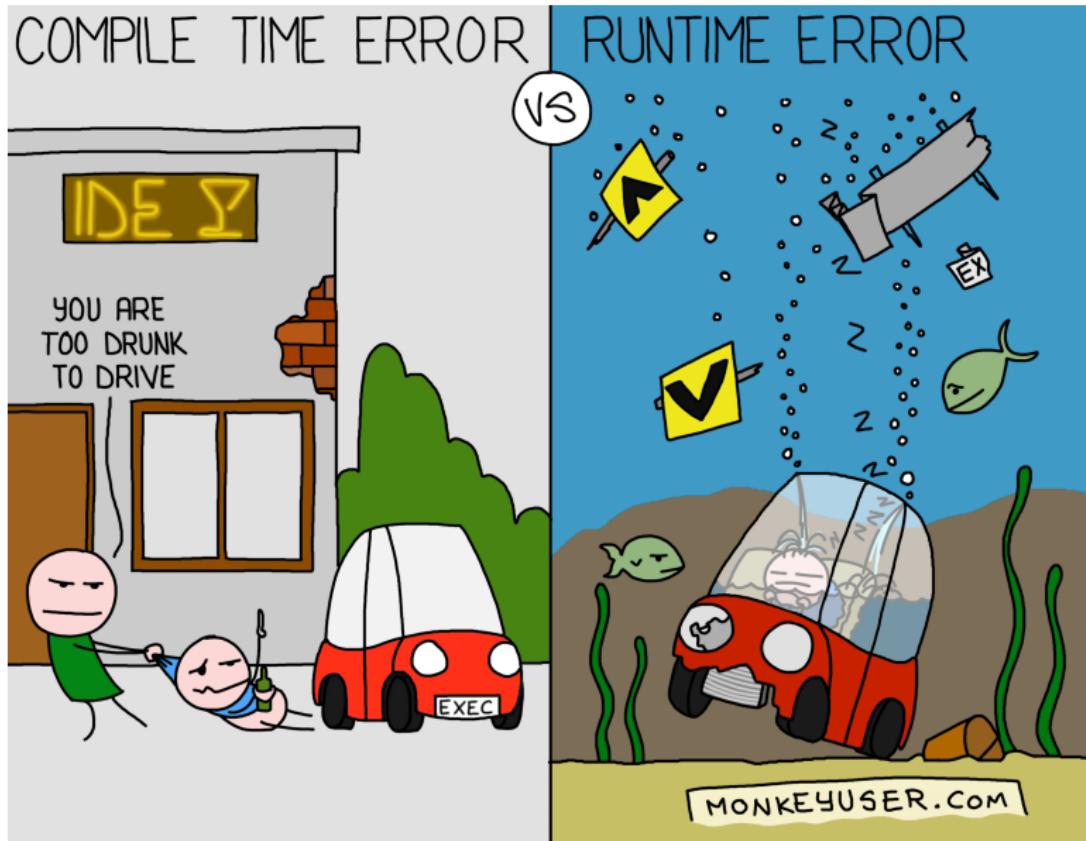
Session types prevent silly mistakes in **communication**.

Session Types

“I will send Bob a Bool, and then I expect and Int from Alice”

- ▶ protocols become code
- ▶ we introduce ordering of communication
- ▶ messages are typed

Session Types: Static vs. Dynamic



Session Types: Static vs. Dynamic

We will always need dynamic verification of our session types

Because RealWorld systems are:

- ▶ opaque
- ▶ written in multiple languages

Reversibility

Goal: make smart decisions when the circumstances change

Reversibility

Leave behind a trail of breadcrumbs



So we can always find our way back

Reversibility

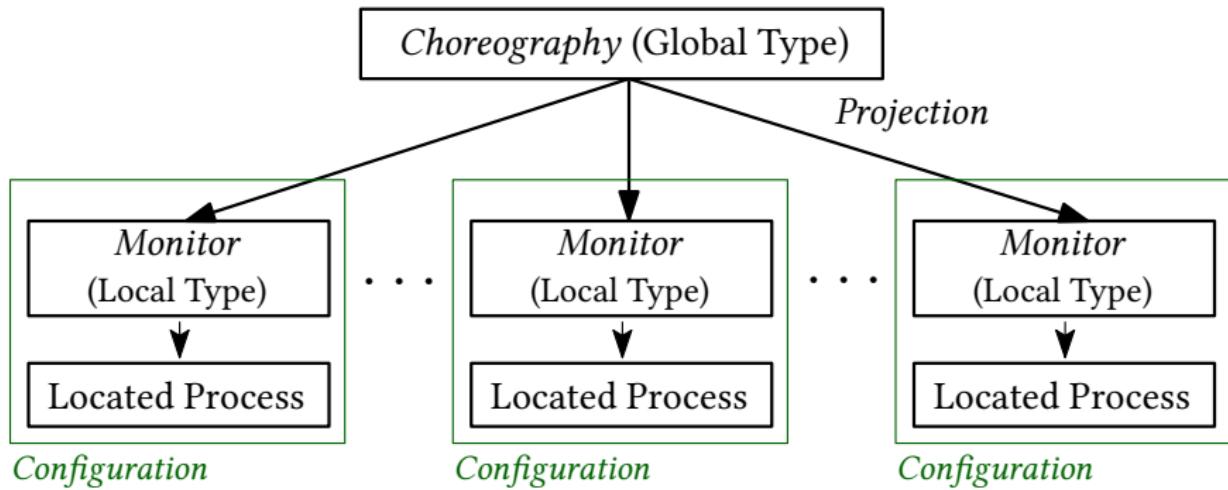
$$\text{backward} \circ \text{forward} \approx \text{identity}$$

Causal Consistency: Reversible steps lead to states that can be reached with forward steps only. No extra new states are introduced.

Implementation

Combining these ideas in practice

Implementing Session Types: Global and Local



Running Example: A Three-Buyer Protocol

Alice (A), Bob (B), and Carol (C) interact with a Vendor (V) :

$$\begin{aligned} G = & A \rightarrow V : \langle \text{title} \rangle. \quad V \rightarrow \{A, B\} : \langle \text{price} \rangle. \\ & A \rightarrow B : \langle \text{share} \rangle. \quad B \rightarrow \{A, V\} : \langle \text{OK} \rangle. \\ & B \rightarrow C : \langle \text{share} \rangle. \quad B \rightarrow C : \langle \{\{\diamond\}\} \rangle. \\ & B \rightarrow V : \langle \text{address} \rangle. \quad V \rightarrow B : \langle \text{date} \rangle.\text{end} \end{aligned}$$

where $\{\{\diamond\}\}$ is a **thunk process**: a type $((\text{()}) \rightarrow \text{Process})$.

Bob sends Carol some code with the protocol; she must activate it.

Implementing Session Types: Use

```
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
```

Implementing Session Types: Use

```
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
```

Derived local type for A:

```
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
```

Implementing Session Types: Definition

```
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
    }
  | RecursionPoint next
  | RecursionVariable
  | WeakenRecursion next
  | End
deriving (Show, Functor)
```

Implementing the Process Calculus: Definition

Make a constructor for everything that we're interested in

```
data ProgramF value next
  -- passing messages
  = Send {...}
  | Receive {...}
  | Parallel next next
  -- choice
  | Offer Participant (List (String, next))
  | Select Participant (List (String, value, next))
  -- other
  | Application Participant Identifier value
  | NoOp
```

Improvements over the standard pi calculus:

- ▶ Process delegation & ownership
- ▶ Asynchronous communication

Implementing the Process Calculus: Use

- ▶ StateT threads ownership through the computation
- ▶ Free provides nice syntax with do-notation

```
bob = do
  thunk <-
    H.function $ \_ -> do
      H.send (VString "accursedUnutterablePerformIO")
      d <- H.receive
      H.terminate
  price <- H.receive
  share <- H.receive
  H.send (VBool True)
  H.send (VBool True)
  H.send share
  H.send thunk

carol = do
  h <- H.receive
  code <- H.receive
  H.applyFunction code VUnit
```

Adding Reversibility: Types

```
data TypeContextF a previous
= Transaction (LocalType.Transaction a previous)
| Spawning Location Location Location 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
| Empty
| R previous
| Wk previous
| V previous
```

Adding Reversibility: Processes

We also need to store

- ▶ used variable names
- ▶ unused branches in select and offer
- ▶ function applications: the function and the argument
- ▶ message values

All of these have their own stack

The Monitor and Synchronization

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

ExecutionState

```
data ExecutionState value =  
  ExecutionState  
    { variableCount :: Int  
    , locationCount :: Int  
    , applicationCount :: Int  
    , queue :: Queue value  
    , participants :: Map Participant (Monitor value String)  
    , locations :: Map Location  
      ( Participant  
      , List OtherOptions  
      , Program value  
      )  
    , isFunction :: value -> Maybe (Identifier, Program value)  
    }
```

Conclusion