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

Problem

Concurrency is hard

Roadmap

Core Concepts: The pi-calculus, session types and reversibility

Moving Forward

Moving Backward

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

1. function creation: `\x -> x`
2. function application: `f x`
3. reduction: `(\x -> x) 42 => 42`

The pi-calculus defines

1. send: $\bar{x}$`<y>.P` send value y over channel x, then run P
2. receive: `x(y).P` receive on channel x, bind the result to y, then run P
3. parallel: `P|Q` run P and Q simultaneously
4. reduction: $\bar{x}$`<a>.P | x(b).Q => P | Q`

Session Types

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

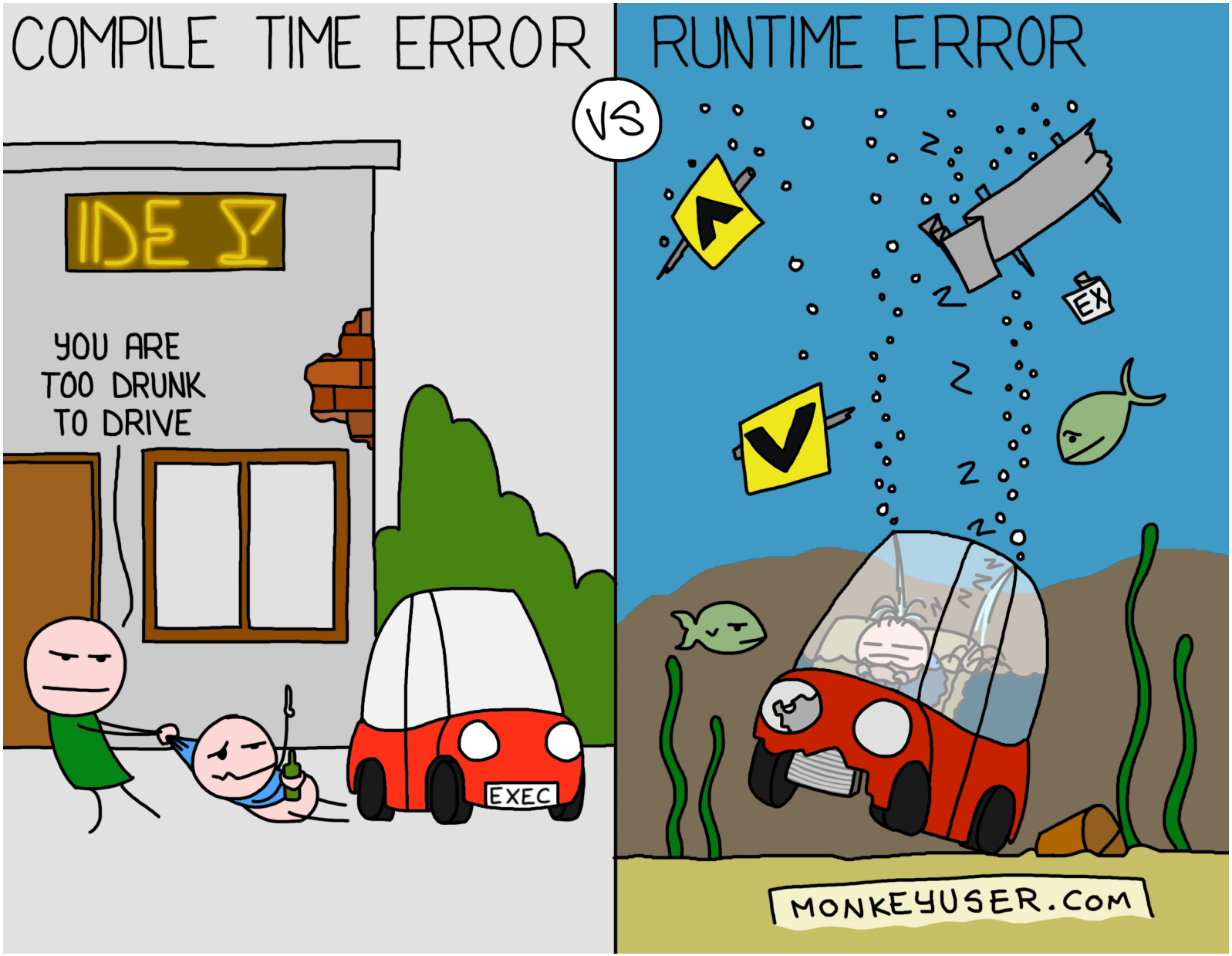
Session types prevent stupid mistakes in **communication**.

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 session types

Because `RealWorld` systems are:

- opaque
- written in multiple languages

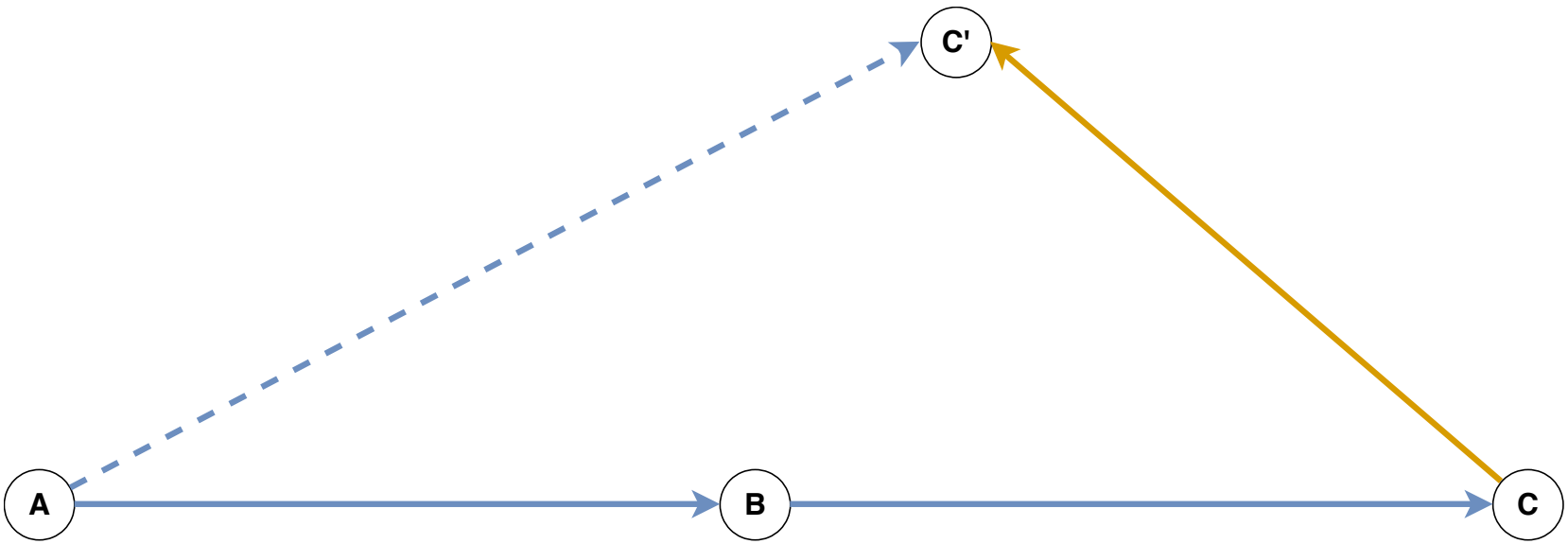
Reversibility

Goal: recover from errors

Reversibility

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

Causal Consistency: Any point we can reach (by going forward and backward) we can also reached by only going forward from the beginning



Reversibility

Leave behind a trail of breadcrumbs

So we can always find our way back

Problem

These three things are cool, can we let them work together in a nice way

in theory: yes, in practice? tbd

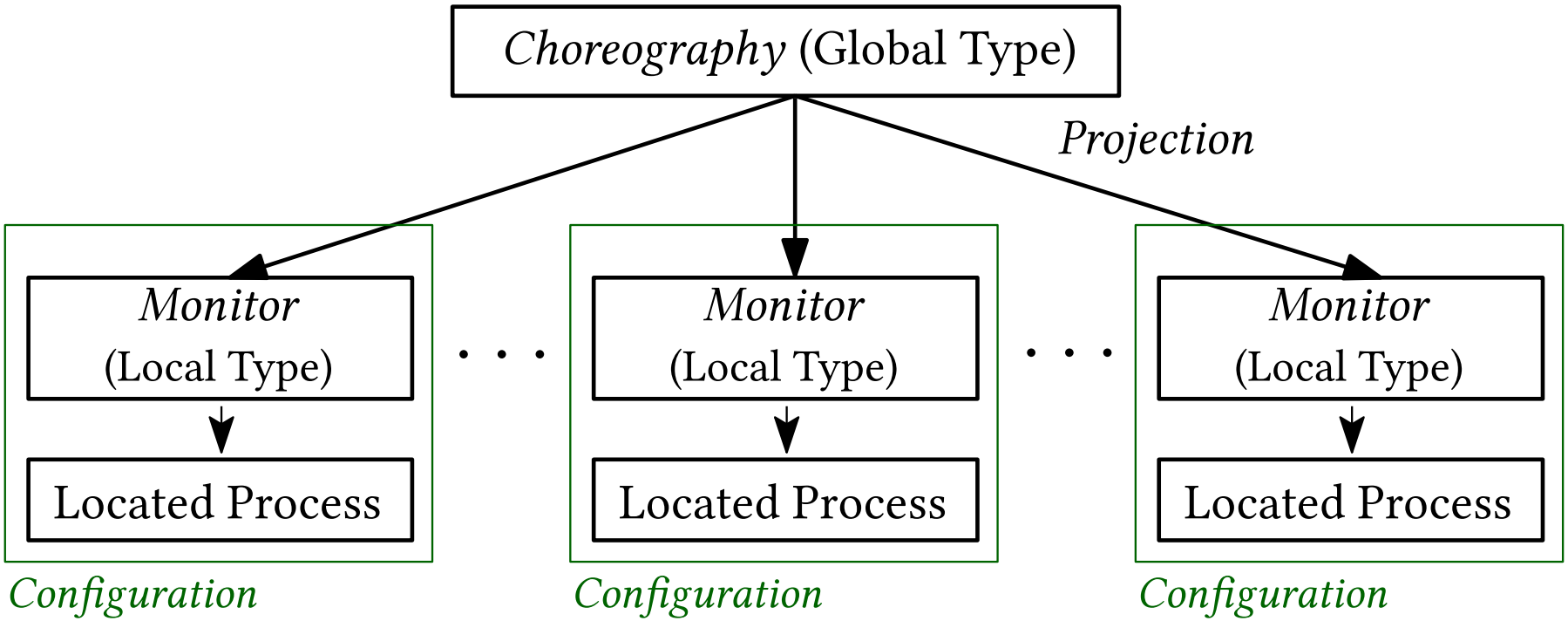
Roadmap

Core Concepts: The pi-calculus, session types and reversibility

Moving Forward

Moving Backward

Implementing Session Types: Global and Local



Running Example: A Three-Buyer Protocol

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

$$
\begin{aligned}
G = \mathtt{A} \to \mathtt{V} : \langle \mathsf{title} \rangle.\ \ \mathtt{V} \to \{\mathtt{A}, \mathtt{B}\} : \langle \mathsf{price} \rangle. \\
\mathtt{A} \to \mathtt{B} : \langle \mathsf{share} \rangle.\ \ \mathtt{B} \to \{\mathtt{A}, \mathtt{V}\} : \langle \mathsf{OK} \rangle. \\
\mathtt{B} \to \mathtt{C} : \langle \mathsf{share} \rangle.\ \ \mathtt{B} \to \mathtt{C} : \langle \{\{\diamond\}\} \rangle. \\
\mathtt{B} \to \mathtt{V} : \langle \mathsf{address} \rangle.\ \ \mathtt{V} \to \mathtt{B} : \langle \mathsf{date} \rangle.\mathtt{end}
\end{aligned}
$$

where $\{\{\diamond\}\}$ is a **thunk process**: a type (`() -> Process`).
Bob sends Carol some code with the protocol; she must activate it.

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 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 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))
    | Application Participant Identifier value
    | NoOp
    -- syntactic sugar for better examples
    | Let Participant Identifier value next
    | IfThenElse Participant value next next
```

Improvements over the standard pi calculus:

- We can send part of a session over a channel
- Communication is decoupled with a queue

Implementing the Process Calculus: Use

Then use the `Free` monad to get nice syntax for free:

```
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
```

Roadmap

Core Concepts: The pi-calculus, session types and reversibility

Moving Forward

Moving Backward

Adding Reversibility: Types

```
data TypeContextF a previous
    = Empty
    | Transaction (LocalType.Transaction a 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
    | Spawning Location Location Location previous
    | R previous
    | Wk previous
    | V previous
    -- sugar
    | Branched { owner :: Participant, continuation :: previous }
    | Assignment { owner :: Participant, continuation :: previous }
```

Adding Reversibility: Processes

We also need to store

- used variable names
- unused branches in select, offer and if-then-else
- 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, Progra
        , isFunction :: value -> Maybe (Identifier, Program value)
        }
```

Conclusion