

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

Bridging the gap between theory and practice

Context

A practical implementation of PPDP 2017 find title

- ▶ validation of theoretical ideas
- ▶ a step toward reversible concurrent debuggers and using failure information

Core Concepts

- ▶ the pi-calculus: a calculus for concurrent computation
- ▶ session types: a type system for concurrent computation
- ▶ reversibility: moving backwards through a program

The lambda-calculus

Our favorite model for sequential computation: The lambda calculus

$M, N ::= x$	Variable
$(\lambda x.M)$	Function definition
$(M N)$	Applying a function to an argument

β -reduction:

$$(\lambda x.M) E \rightarrow (M[x/E])$$

There is one type per term

The pi-calculus

In contrast, the pi-calculus defines

$P, Q, R ::= \bar{x}\langle y \rangle.P$	Send the value y over channel x , then run P
$ \quad x(y).P$	Receive on channel x , bind the result to y , then run P
$ \quad P Q$	Run P and Q simultaneously
$ \quad (\nu x)P$	Create a new channel x and run P
$ \quad !P$	Repeatedly spawn copies of P
$ \quad 0$	Terminate the process
$ \quad P + Q$	(Optionally) Nondeterministic choice

reduction:

$$\bar{x}\langle z \rangle.P | x(y).Q \rightarrow P|Q[z/y]$$

One type per channel, possibly many involving a term.

Session Types

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

Session Types

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

- ▶ Send without a receive
- ▶ Receive without a send
- ▶ Sending/Receiving before you're supposed to

An idea introduced in the 90s by ...

Session Types

Global Type

- ▶ There is a Transaction between Carol and Bob, a Bool is sent.
- ▶ There is a Transaction between Alice and Carol, an Int is sent.

Session Types

Global Type

- ▶ There is a Transaction between Carol and Bob, a Bool is sent.
- ▶ There is a Transaction between Alice and Carol, an Int is sent.

Local Type (for Carol)

- ▶ I will send Bob a Bool
- ▶ I expect an Int from Alice

Session Types

Global Type

- ▶ There is a Transaction between Carol and Bob, a Bool is sent.
- ▶ There is a Transaction between Alice and Carol, an Int is sent.

Local Type (for Carol)

- ▶ I will send Bob a Bool
- ▶ I expect an Int from Alice

Properties:

- ▶ Order: enforced ordering of communication over a channel
- ▶ Safety: sent values are typed, there can be no mismatch
- ▶ Progress: every sent message is eventually received

Running Example: A Three-Buyer Protocol

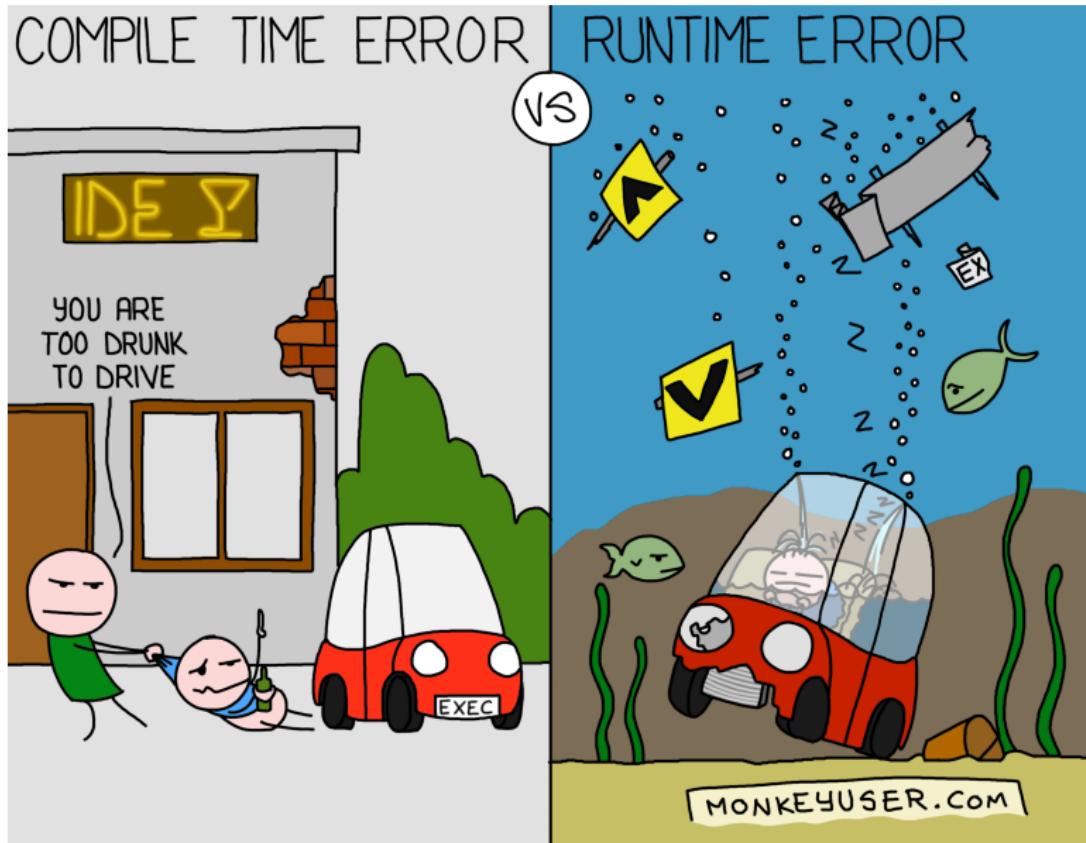
Alice (A), Bob (B), and Carol (C) interact with a Vendor (V) :

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

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

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

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

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

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

Implementation

Combining these ideas in practice

Implementing Session Types: Global and Local

We aim to implement:

```
type Session a = StateT ExecutionState (Except Error) a
forward :: Location -> Session ()
backward :: Location -> Session ()
```

Where

- ▶ Session models our computation
- ▶ ExecutionState contains our types and programs
- ▶ Except Error provides a way to fail
- ▶ Location models threads or machines

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:

- ▶ Protocol 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

```
decision <- H.receive  
H.send decision  
H.send decision
```

- ▶ unused branches in select and offer

```
data OtherOptions  
  = OtherSelections (Zipper (String, Value, Program Value))  
  | OtherOffers (Zipper (String, Program Value))
```

- ▶ function applications: the function and the argument

```
Map Identifier (Value, Value)
```

Adding Reversibility: Processes

- ▶ message values

History Stack



Receive

Roll Receive

Roll Send

Queue

Front	x	y	z
42	x	y	z

x	y	z
---	---	---

42	x	y	z
----	---	---	---

x	y	z
---	---	---

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