Distributed Systems Extensions for the Dunai FRP Library

https://github.com/jgotoh/distributed-frp-dunai

Julian Götz

Leipzig, 3. März 2020

Hochschule für Technik, Wirtschaft und Kultur Leipzig

1. Distributed Systems
2. Functional Reactive Programming
3. Dunai
4. Distributed Systems Extensions for the Dunai FRP Library
5. Cloud Haskell
6. Implementation
7. Main Loop
8. Next steps

Distributed Systems

According to Andrew S. Tanenbaum and Maarten Van Steen[TS07]:

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

- collaborating autonomous computers
- appear as one unit
- Multiplayer games: playing together in the same virtual world

Functional Reactive Programming

- Paradigm for implementing hybrid systems [Hud+02]
- Values that change over continous time: Signals

```
type Signal a = Time -> a
```

- Discrete Signals: Events
- Arrowized FRP: Signal Functions

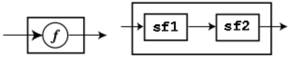
```
type SF a b = Signal a -> Signal b
```

  - SFs are Arrows: `Arr a b` [Hug00]
  - primitives and combinators
  - `sf >>> sf1 &&& sf2`
  - implicit Signals to avoid Space-Time Leaks
- FRP program: composition of a set of signals into one composite signal

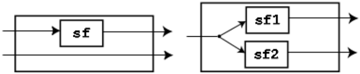
[NCP02]

Functional Reactive Programming

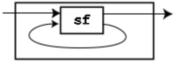
- `arr :: (a -> b) -> SF a b`
- `(>>>) :: SF a b -> SF b c -> SF a c`
- `first :: SF a b -> SF (a,c) (b,c)`
- `(&&&) :: SF a b -> SF a c -> SF a (b,c)`
- `loop :: SF (a,c) (b,c) -> SF a b`



(a) `arr f` (b) `sf1 >>> sf2`



(c) `first sf` (d) `sf1 &&& sf2`



(e) `loop sf`

Dunai

- Parametrized SFs with monads [Dom18]

```haskell
newtype MSF m a b
```

- `f :: Monad m =>m a`
- tries to subsume all FRP libraries
- Computation in a context with a result value of type a
- new primitive
  `arrM :: Monad m =>(a -> m b) -> MSF m a b`
- BearRiver is the successor of Yampa implemented using Dunai
  - `type ClockInfo m = ReaderT DTime m`
  - `type SF m = MSF (ClockInfo m)`

Distributed Systems Extensions for the Dunai FRP Library

Implementation of a library

- facilitate integration of autonomous machines running FRP into a single coherent system
- Communication via Client/ Server architecture
- Synchronisation of FRP states to migitate impact of network
- Clients send CommandPackets (input)
- Servers send UpdatePackets (state) [Val19]

Cloud Haskell

- distributed computing framework inspired by Erlang [AVW93], designed by Epstein, Black and Peyton Jones in 2011 [EBP11]
- processes communicate via messages
- as opposed to shared data
- implemented in set of packages called distributed-process [CWV18]
- agnostic to transport layer (TCP, UDP, ...)
- built-in support for Client/ Server

Processes and Nodes

- Processes
    - Erlang: virtual machines that evaluate Erlang functions [Arm07, p. 141]
    - Haskell: computations in the Process Monad
    - identified by unique process identifiers
    - lightweight creation, destruction and low scheduling overhead
- . . . running on Nodes
    - runtime system executing Erlang/ Haskell code
    - able to communicate with other Nodes
    - identifiable by a unique NodeId

Processes and Nodes

- Erlang: `Pid = spawn(Fun)`
- Haskell:

  `spawn :: NodeId -> Closure (Process()) -> Process ProcessId`

Messages

- Messages
- Erlang: send asynchronously: `Pid ! Message`
- Cloud Haskell: message consists of binary data and type representation

```
send :: Serializable a =>ProcessId -> a -> Process ()
class (Binary a, Typeable a) =>Serializable a
instance (Binary a, Typeable a) =>Serializable a
```

- siehe `distributed-paddles/GameState.hs`

Pattern Matching on incoming messages

Erlang:

```
math() ->
  receive
    {add, Pid, Num1, Num2} ->
      Pid ! Num1 + Num2;
    {divide, Pid, Num1, Num2} when Num2 != 0 ->
      Pid ! Num1 / Num2;
    {divide, Pid, _, _} ->
      Pid ! div_by_zero
  end,
  math().
```

Pattern Matching on incoming messages

Cloud Haskell:

```
data Add = Add ProcessId Double Double
data Divide = Divide ProcessId Double Double
data DivByZero = DivByZero
% Serializable instances are omitted
math :: Process ()
math =
  receiveWait
    [ match (\(Add pid num1 num2) ->
          send pid (num1 + num2)),
      matchIf (\(Divide _ _ num2) -> num2 /= 0)
          (\(Divide pid num1 num2) ->
              send pid (num1 / num2)),
      match (\(Divide pid _ _) ->
          send pid DivByZero) ]
  >> math
```

Typed Channels

- exclusive to Cloud Haskell
- uses Haskell's static type system
- ensures only messages of a specific type are sent to a process
- `type Channel a = (SendPort a, ReceivePort a)`

```
newChan :: Serializable a =>
    Process (SendPort a, ReceivePort a)
receiveChan :: Serializable a =>
    ReceivePort a -> Process a
sendChan :: Serializable a =>
    SendPort a -> a -> Process ()
mergePortsBiased :: Serializable a =>
    [ReceivePort a] -> Process (ReceivePort a)
mergePortsRR:: Serializable a =>
    [ ReceivePort a] -> Process (ReceivePort a)
```

Fault Tolerance

- Processes can be observed whether they terminate unexpectedly and expectedly
- monitor (unidirectional)
  - Erlang: `MonitorRef = erlang:monitor(process, Pid2)`
  - Haskell: `monitor :: ProcessId -> Process MonitorRef`
- link (bidirectional)
  - Erlang: `link(Pid2)`
  - Haskell: `link :: ProcessId -> Process ()`
- monitoring/ linking of Nodes and Channels is also possible

Dynamic Code Update

- Erlang: compiled to bytecode, run by interpreter
- Haskell: compiled to machine code, run by OS

```
-module(m).
-export([loop/0]).

loop() ->
  receive
    code_switch ->
      m:loop();
    Msg ->
      ...
      loop()
  end.
```

Implementation

- Network communication via Cloud Haskell (Process monad)
- Dunai runs in IO
- Cloud Haskell's messaging system only works in context of Process
- Dunai and Cloud Haskell need to communicate (UpdatePackets, Commandpackets)
- shared data structures are necessary: STM[Har+08]
    - lockfree synchronisation via revertible transactions

Main Loop

- Dunai: `reactimate :: MSF m () () -> m ()` [Dom18]
- BearRiver: [Dom18]

```
reactimate :: Monad m
=> m a                                   -- initial sense
-> (Bool -> m (DTime, Maybe a))  -- sense
-> (Bool -> b -> m Bool)           -- actuate
-> SF Identity a b
m ()
```

Main Loop

- reactimateClient (simplified):

```
reactimateClient :: Monad m
=> m (DTime, Maybe a) -- sense
-> (b -> m()) -- actuate
-> SF Identity (a, Maybe netin) b -- pure Signal Functions
-> m (Maybe netin) -- read UpdatePackets
-> ((DTime, Maybe a) -> m ()) -- send CommandPackets
-> m ()
```

- reactimateServer (simplified):

```
reactimateServer :: Monad m
=> m (DTime, Maybe a) -- sense
-> (b -> m()) -- actuate
-> SF Identity (a, Maybe netin) b -- pure Signal Functions
-> m (Maybe netin) -- read CommandPackets
-> (b -> m ()) -- send UpdatePackets
-> m ()
```

Next steps

- Synchronisation
    - Clock Synchronisation
    - State Synchronisation
        - Dead Reckoning
        - Error correction (Time Warp)

Quellen I

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Raleigh, NC, und Dallas, TX: The Pragmatic Programmers, LLC, 2007. ISBN: 978-1-9343560-0-5. URL: http://www.pragprog.com/titles/jaerlang/programming-erlang.

[AVW93] Joe Armstrong, Robert Virding und Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993. ISBN: 978-0-13-285792-5.

[CWV18] Duncan Coutts, Nicolas Wu und Edsko de Vries. *distributed-process: Cloud Haskell: Erlang-style concurrency in Haskell*. 2018. URL: https://hackage.haskell.org/package/distributed-process (besucht am 10. 01. 2020).

Quellen II

[Dom18] Iván Pérez Domínguez. “Extensible and Robut Functional Reactive Programming”. Diss. Nottingham, UK, 2018.

[EBP11] Jeff Epstein, Andrew P. Black und Simon L. Peyton Jones. “Towards Haskell in the cloud”. In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Hrsg. von Koen Claessen. ACM, 2011, S. 118–129. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034690. URL: https://doi.org/10.1145/2034675.2034690.

[EH97] Conal Elliott und Paul Hudak. “Functional Reactive Animation”. In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ICFP ’97. Amsterdam, The Netherlands: ACM, 1997, S. 263–273. ISBN: 0-89791-918-1. DOI: 10.1145/258948.258973. URL: http://doi.acm.org/10.1145/258948.258973.

Quellen III

[Eri19] Ericsson AB. *Erlang/ OTP System Documentation*. 2019. URL: https://erlang.org/doc/pdf/otp-system-documentation.pdf (besucht am 14. 01. 2020).

[Har+08] Tim Harris u. a. "Composable memory transactions". In: *Commun. ACM* 51.8 (2008), S. 91–100. DOI: 10.1145/1378704.1378725. URL: https://doi.org/10.1145/1378704.1378725.

Quellen IV

[Hud+02] Paul Hudak u. a. “Arrows, Robots, and Functional Reactive Programming”. In: *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*. Hrsg. von Johan Jeuring und Simon L. Peyton Jones. Bd. 2638. Lecture Notes in Computer Science. Springer, 2002, S. 159–187. ISBN: 3-540-40132-6. DOI: 10.1007/978-3-540-44833-4_6. URL: https://doi.org/10.1007/978-3-540-44833-4%5C_6.

[Hug00] John Hughes. “Generalising monads to arrows”. In: *Sci. Comput. Program.* 37.1-3 (2000), S. 67–111. DOI: 10.1016/S0167-6423(99)00023-4. URL: https://doi.org/10.1016/S0167-6423(99)00023-4.

Quellen V

[NCP02] Henrik Nilsson, Antony Courtney und John Peterson. "Functional Reactive Programming, Continued". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. 565022. Pittsburgh, Pennsylvania: ACM, 2002, S. 51–64. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581695. URL: http://doi.acm.org/10.1145/581690.581695.

[PBN16] Ivan Perez, Manuel Bärenz und Henrik Nilsson. "Functional Reactive Programming, Refactored". In: *Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. Nara, Japan: ACM, 2016, S. 33–44. ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976010. URL: http://doi.acm.org/10.1145/2976002.2976010.

Quellen VI

[TS07] Andrew S. Tanenbaum und Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007. ISBN: 978-0-13-239227-3.

[Val19] Valve Corporation. *Source Multiplayer Networking*. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. 2019. (Besucht am 13. 12. 2019).