The Monad.Reader/Issue3/Join Hs

From HaskellWiki

< The Monad.Reader | Issue3

Contents

1 Concurrent and Distributed Programming with JoinHs

by Einar Karttunen for The Monad Reader Issue Three, (date 2005-05-31T14:11:05Z)

JoinHs is a new experimental dialect of Haskell supporting concurrent and distributed programming using Join-calculus (see "The reflexive chemical abstract machine and the join-calculus", by CÈdric Fournet and Georges Gonthier, POPL'96) as a basis. The result is a declarative way to define concurrent abstractions, which can be distributed in a transparent fashion.

The system is based on unidirectional asynchronous channels with fixed receiving points. These join-points declare new channels and specify how they are synchronized with each other. Channels are represented as polyadic functions in the IO monad. To get a bidirectional channel with replies one simply supplies a channel with a continuation argument (which should be a channel) where the result is put. The library supports syntactic sugar for this.

Join-patterns define a set of channels and declare how they are synchronized. The form of a join-pattern is

```
join c1 | ... | cn = P1
 ci | ... | cm = P2
 ...
```

Each expression *P* is executed when all the channels on the left side have a message ready.

Join-patterns are compiled into concurrent Haskell by a simple preprocessor, which is available alongside the library. The system works currently with GHC6.4 and can be obtained from the homepage (http://www.cs.helsinki.fi/u/ekarttun/JoinHs/) .

1.1 A Simple Counter

Our first example is a simple counter with two operations - get and inc. The pattern can be read as follows: when both *count* and *inc* channel have pending data evaluate *count (n+1) >> reply* and when both *count* and *get* channels have data evaluate *count n >> reply n*. There is only one message in the *count* channel at any given time, which assures mutual exclusion.

```
createCounter = do
 join count n | inc reply = count (n+1) >> reply
 count n | get reply = count n >> reply n
 count 0
 return (inc,get)

main = do
 (i1,g1) <- createCounter
 let is1 = sync i1
 gs1 = sync g1
 is1 >> is1 >> is1 >> is1
 print =<< gs1
```

The example can be loaded into GHCi with `ghci -F -pgmF joinhsc counter.hs`. More complicated examples may need `-fallow-overlapping-instances` with GHC6.4.

sync is a method for transforming a channel taking a continuation (the channel used for reply) into one returning the value passed to the reply continuation. If the reply-continuation is polyadic then the result is converted into a tuple.

1.2 Introducing distributed computation

For distributed communication the peers need to get to know each other via some service. JoinHs uses a nameserver for this purpose. Nodes can register channels in the nameserver and lookup who has registered a given name. The nameserver deregisters names when nodes shut down.

As a simple example, we look at a server exporting the Fibonacci function and clients using it.

```
import Join

main = do
 join fibo n reply = if n <= 1 then reply 1
 else do v1 <- sync $ fibo (n-1)
 v2 <- sync $ fibo (n-2)
 reply (v1+v2)
 register "fibo" (fibo :: Int -> Sync Int)
```

```
eternalSleep
```

This uses channels to calculate every recursive call of Fibonacci and finally replies to the supplied continuation. The explicit type is used to make the registered function monomorphic. Sync is just a helper type in the library to make the signatures look prettier. It

could be defined thus:

```
type Sync a = (a -> IO ()) -> IO ()
```

The corresponding client is quite simple:

```
import Join

getFibo :: IO (Int -> Sync Int)
getFibo = Join.lookup "fibo"

main = do
 fibo <- getFibo
 print =<< sync (fibo 10)
 print =<< sync (fibo 15)
 print =<< sync (fibo 20)
```

To run the example (which is located under `examples/fibo` in the JoinHs distribution) first start the nameserver, with the command "jhnsd". Then run "fibo" and "client", which should just print the results. If a program is run on a different machine than the one running the nameserver then the environment variable JOINHS_NS is used as the hostname of the server. The library has an API for using nameservers with a specified host and port - or even multiple nameservers at the same time.

1.3 Implementation

The preprocessor compiles join-patterns into concurrent Haskell using MVars and pattern matching. Asynchronous execution is assured by using threads, which are very cheap with GHC.

1.3.1 Channel Representation

Channels are currently represented as polyadic functions in the IO monad. This makes using them most natural in Haskell, but makes introspection hard. Synchronous channels are represented with an additional argument that is the channel to send the return value to. The runtime library offers a combinator called sync, which takes a channel and creates a suitable return channel for it. The tricky part is that the return channel may be polyadic - but Haskell allows only single values to be returned - thus a conversion to a tuple is needed.

Another possible form of representing the channels would be to make them accept only one argument as a tuple. This would make the implementation much simpler, but would not be very natural in Haskell.

A third possibility is to represent the channels abstract data types parameterized by a tuple (or a HList) of the arguments and provide methods to send data or convert to a (polyadic) function representation. This would allow introspection easily, but the transformation from a tuple to a polyadic function seems quite hard (although it should be possible given some time/caffeine).

1.3.2 Implementing distributed channels

When sending a channel to a remote side a proxy is registered for the connection which decodes arguments and sends them into the local channel. A reference to the proxy is sent to the other side.

When a reference to a remote channel is received a local proxy channel is created. This channel receives arguments, encodes them and sends them to the remote side. As synchronization only happens at the join-pattern definition sites it is of no concern to the distribution.

1.4 Related Work

Most of the ideas come from Join-calculus and the JoCaml-language (http://pauillac.inria.fr/jocaml/) . JoCaml is based on OCaml and supports mobile code in addition to channels. This is not supported at all in JoinHs. Mobile code is good in a parallel setting with trust, but without trust it needs a sandbox. Much more research would be needed in order to support this in Haskell - and probably using bytecode instead of native compiled code.

A previous attempt of distributed communication in Haskell was "Haskell with Ports" (see "Distributed Programming in Haskell with Ports", by Frank Huch and Ulrich Norbisrath). It provided ports which allowed their creating process to read and write them - and all other processes to write them. The semantics and in particular synchronization were more *ad hoc* than in JoinHs. Also channels in Haskell with ports were not polyadic. On the other hand Haskell with Ports is much more mature and currently handles failures better.

Erlang (http://www.erlang.org) is a pure dynamically typed functional language, offering massive concurrency and distribution. In Erlang processes communicate by sending messages to each other. Addressing is based on process ids. The built-in distribution mechanism is very mature and offers very good integration of nodes. However it is not designed for untrusted peers and does not have the benefits of static typing. However serializing values and sending them on wire between peers has been very easy and alluring...

1.5 Future Work

Currently JoinHs uses Show and Read typeclasses for serialization for simplicity. In future it is planned to switch to SerTH (http://www.cs.helsinki.fi/u/ekarttun/SerTH) to get benefits of a fast binary serialization framework supporting cyclic structures.

Adding strong encryption to the on-wire representation should not be very hard. The tentative plan is to register keys of nodes in the nameservers together with functions so

the clients need only to know the key of the nameserver or trust it.

A better mechanism for handling failures is needed, which will probably be tied to an alternative representation of channels as abstract datatypes.

Garbage collecting channels turns out to be surprisingly easy as at all times one needs only to consider single pairs of peers. It should suffice to add a finalizer to the local proxy handling a remote channel messaging the remote side that the channel has died.

Retrieved from "http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue3/Join_Hs"

Category: Article
