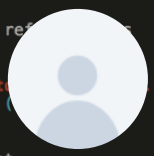
```haskell
-- | Assemble the RSA pipeline.
rsaPipeline :: RunInformation -> Effect IO ()
rsaPipeline runInfo = fetchRawInputs runInfo
                  >-> preprocessInputs
                  >-> addForecasts
                  >-> createAndWriteReport runInfo

-- | Grab raw inputs from the database.
fetchRawInputs :: RunInformation -> Producer RawInputs IO ()
fetchRawInputs (_, reportId, runId) = do
  mongoPipe <- lift . connectToDatabase $ rsaConfig^.db.dbHost
  keys      <- lift . runDbAction mongoPipe . handleErr $ getRunKeys runId
  rawInputs <- lift . runDbAction mongoPipe $ getRawInputs reportId keys
  lift . close $ mongoPipe
  yield rawInputs

-- | Preprocess raw inputs and yield them downstream.
preprocessInputs :: Monad m => Pipe RawInputs Inputs m ()
preprocessInputs = do
  rawInputs <- await
  let refinedInputs = preprocess rawInputs
  yield refinedInputs

-- | Augment inputs with participant forecasts and yield them downstream.
addForecasts :: Monad m => Pipe Inputs Inputs m ()
addForecasts = do
  refinedInputs <- await
  let inputs = refinedInputs&participants .~ assignForecasts ref
  yield $!! inputs

-- | Consume inputs, create a report document, and write it t
createAndWriteReport :: RunInformation -> Consumer Inputs IO (
createAndWriteReport (reportType, _, runId) = do
  inputs    <- await
  mongoPipe <- lift . connectToDatabase $ rsaConfig^.db.dbHost
  when (reportType == PDOR) $
    lift . runDbAction mongoPipe $ writePdorToDatabase runId inputs
  lift . close $ mongoPipe
```

Haskell and Rust.



Posted by Chris Allen - 26 November, 2018

FP Complete is known for our best-in-class DevOps automation tooling in addition to Haskell. We use industry standards like Kubernetes and Docker. We're always looking for new developments in software that help us empower our clients.

Rust is an up-and-coming programming language whose history starts in 2010. Rust has much overlap in its strengths with Haskell. We are impressed with Rust's tooling, library ecosystem, and the community behind all of it. We're also keenly interested in incorporating Rust into our work more and more.

Haskell has served FP Complete very well throughout its history, but it isn't ideal in all circumstances. Haskell works exceptionally well as an application language, especially for web applications and networked servers. Haskell has seen productive use in everything from financial technology to non-profit web platforms. We believe Haskell excels when you want to be able to maintain quality and maintainability without compromising developer productivity.

Haskell and Rust are the same

Haskell and Rust have shared goals and design priorities. Those overlapping priorities align well with what we value at FP Complete:

- Better, cheaper, more automatic correctness assurances through types and tooling. Both Haskell and Rust have sum types, polymorphism, type inference, type classes (Rust's traits), associated types, and ~~accidental~~ turing-completeness in their type system. Both languages are immutable-by-default and avoid mutation of shared references. Concurrent programs, in particular, are less costly and easier to get right in Haskell and Rust.
- Quality assurance by leveraging multiple software testing methodologies.
- Maintaining a strong performance and concurrency story so that your prototypes can be extended and built upon rather than binned and replaced.
- See our post about whether Rust is functional to see how Haskell and Rust compare in that dimension.

Sum types

Here's an example of structures being alike in Haskell and Rust:

- Haskell:

```
data Maybe a =
    Nothing
  | Just a

defaultOne :: Maybe Int -> Int
defaultOne Nothing = 1
defaultOne (Just n) = n
```

- Roughly the same in Rust:

```
pub enum Option<T> {
    None,
    Some(T),
```

```
}

pub fn default_one(v: &Option<i64>) -> i64 {
    match v {
        None => 1,
        Some(n) => n.clone(),
    }
}
```

You should uppercase type variables when writing Rust. In Haskell, they're always lowercase. Lifetimes start with a single quote and are lowercase in Rust.

Polymorphism

```
maybeEither :: Maybe a -> Either String a
maybeEither Nothing = Left "The value was missing!"
maybeEither (Just v) = Right v
```

```
pub fn optional_result<T>(v: Option<T>) -> Result<T, String> {
    match v {
        None => Err("The value was missing!".to_string()),
        Some(x) => Ok(x),
    }
}
```

Here we didn't need to know the type of the values inside the `Maybe` or `Option` type. Instead, we left them polymorphic.

Type classes or traits

I pulled this example from the second edition of The Rust Programming Language:

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct Article {
    pub headline: String,
    pub location: String,
    pub author: String,

    pub content: String,
```

```
}

impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.loca
    }
}
```

Here's my version of the example in Haskell:

```
import Text.Printf

class Summary a where
  summarize :: a -> String

data Article =
  Article {
    headline :: String
  , location :: String
  , author :: String
  , content :: String
  }

instance Summary Article where
  summarize (Article headline location author _) =
    printf "%s, by %s (%s)" headline author location
```

Associated types

A vacuous example to demonstrate the facility:

```
class Hello a where
  type Return
  helloWorld :: a -> Return
```

```
trait Hello {
    type Return;
    fn hello_world(&self) -> Self::Return;
}
```

Haskell and Rust are different

Haskell is going to be stronger when you need maximum productivity when going from a prototype to a production-ready system that can nimbly handle functional and infrastructural changes. Rust can be a better choice when you can plan a bit more and are willing to sacrifice some productivity for better performance or because your project requires a fully capable systems language.

- GHC Haskell has green threads built into the runtime system. Green threads make it much easier to write concurrent programs that "just work." GHC's green-threaded runtime is a large amount of code and is not always the fastest way to do things. GHC's runtime works well for the common-case: web servers and networked applications. GHC can fall short of being ideal elsewhere.

- Rust avoids a runtime entirely and does all of this at a library level. The most common library for solving the c10k problem in Rust is tokio. Here's an example of a simple echo server in Rust using `tokio`:

```
extern crate tokio;

use tokio::prelude::*;
use tokio::net::TcpListener;
use tokio::io::copy;

pub fn main() -> Result<(), Box<std::error::Error>> {
    let addr = "127.0.0.1:3000".parse()?;
    let listen_socket = TcpListener::bind(&addr)?;
    let server = listen_socket
        .incoming()
        .map_err(|e| eprintln!("Error accepting socket: {}", e))
        .for_each(|socket| {
            let (reader, writer) = socket.split();
            let handle_conn =
                copy(reader, writer)
                .map(|copy_info| println!("Finished, bytes copied: {:?}
                .map_err(|e| {
                    eprintln!("Error echoing: {}", e);
                })
                ;
            tokio::spawn(handle_conn)
        })
        ;
    tokio::run(server);
    Ok(())
}
```

Now the same in Haskell:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.9 script

{-# LANGUAGE OverloadedStrings #-}

module Echo where

import Data.Streaming.Network (bindPortTCP)
import qualified Network.Socket as N
import qualified Network.Socket.ByteString as NB
import Control.Concurrent (forkIO)
import Control.Exception (bracket)
import Control.Monad (forever)

main :: IO ()
main = bracket
  (bindPortTCP 3000 "127.0.0.1")
  N.close
  $ \listenSocket -> forever $ do
    (socket, _addr) <- N.accept listenSocket
    forkIO $ forever $ do
      bs <- NB.recv socket 4096
      NB.sendAll socket bs
```

- You must work through more noise dealing with these details in Rust, but you get more fine-grained control in exchange.

Here's an example in Haskell using forkIOWithUnmask:

```
-- From https://www.fpcomplete.com/blog/2018/04/async-exception-handlin
import Control.Concurrent
import Control.Exception
import System.IO

main :: IO ()
main = do
  hSetBuffering stdout LineBuffering
  putStrLn "Acquire in main thread"
  tid <- uninterruptibleMask_ $ forkIOUnmask $ \unmask ->
    unmask (putStrLn "use in child thread" >> threadDelay maxBound)
      `finally` putStrLn "cleanup in child thread"

  killThread tid -- built on top of throwTo
```

```
  putStrLn "Exiting the program"
```

GHC Haskell's runtime also lets you cancel long running CPU tasks, which is notoriously tricky elsewhere. `throwTo` and `killThread` are the most common means of doing so.

Doing something equivalent in Rust requires writing cooperative threading behavior into the threads you want to be able to kill:

```
use std::thread;
use std::time::Duration;
use std::sync::mpsc::{self, TryRecvError};
use std::io::{self, BufRead};

fn main() {
    println!("Acquire in main thread");
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        loop {
            println!("use in child thread");
            // You can't do blocking operations like this in Rust.
            // It won't reach the `rx.try_recv()` or match
            // thread::sleep(Duration::from_millis(std::u64::MAX));
            match rx.try_recv() {
                Ok(_) | Err(TryRecvError::Disconnected) => {
                    println!("Terminating.");
                    break;
                }
                Err(TryRecvError::Empty) => {}
            }
        }
    });
    let _ = tx.send(());
    println!("Exiting the program");
}
```

However, this cooperation means you can't ever block indefinitely in your Rust code, or your thread could stay stuck for the entire lifespan of your process. In GHC Haskell, your code automatically yields control to the runtime whenever it allocates memory. The strength of Haskell here is that you get the ability to preempt or kill threads without doing unspeakable violence to your code's control flow.

Rust has some hard design limitations which are reasonable for what it prioritizes. Haskell, Java, Python, Ruby, and JS also have garbage collection. Rust does edge into the same territory, but Rust has a specific mission to be a better systems language. Targeting core systems applications means Rust is more directly

comparable to C and C++. The Rust core team does their utmost to incorporate modern programming language design. The phrase the Rust team members like to use is that they're trying to make the best 90's era programming language they can. I think this is maybe under-selling it a little, but it's a far sight better than the 60s and 70s vintage PL design available elsewhere.

Rust is stronger for systems programming, embedded, game development, and high-performance computing. Rust is more reliably performant than Haskell, relying less on compiler magic and more on zero-cost abstractions. This emphasis means that the designers try to introduce as much programming convenience as possible where it won't involuntarily reduce performance. Rust's Iterators is an excellent example of this. Haskell tries to obtain some of these benefits with the use of developer-written rewrite rules which are notoriously brittle and hard to debug.

The juice is worth the squeeze

The learning curve of both Haskell and Rust is worthwhile. They are both platforms that you can invest deeply into for robust infrastructure and applications that perform well. On top of that, their respective type systems and idioms enable developers to move faster once they're comfortable.

Both are great choices

Both languages and their associated ecosystems can make normal software development dramatically more tractable and scalable. If you're interested in Haskell or Rust, please check out some of our other blog posts on these great platforms:

- Summary of Haskell
- Summary of Rust
- Our blog posts on Haskell
- Our blog posts on Rust

If you'd like help evaluating Haskell, Rust, or other technologies such as Kubernetes and Docker, please contact us! We are capable of taking your software projects from planning and scoping through project management, implementation, operations, and maintenance.

Contact us!

Topics: Haskell Software, commercial Haskell, Haskell community, rust, GHC, Haskell, rust programming language, concurrency, immutability