AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa inżynierska

Obliczenia rozproszone w języku Haskell
Distributing tasks with Haskell

Autor: *Konrad Lewandowski*
Kierunek studiów: *Informatyka*
Opiekun pracy: *dr inż. Piotr Matyasik*

Kraków, 2018

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Serdecznie dziękuję wszystkim, którzy mnie wspierali i motywowali do działania w czasie trudnego okresu studiów

Spis treści

Listingi kodu

1. Wprowadzenie

W czasach kiedy przewidywania Gordona Moore'a dotyczące dalszego wzrostu mocy obliczeniowej pojedynczych komputerów przestają się sprawdzać, coraz częściej wykorzystujemy metody wykonywania programów oparte na jednoczesnym przetwarzaniu rozproszonym na wielu połączonych ze sobą maszynach. Pomimo pozornej prostoty takiego rozwiązania istnieje bardzo niewiele narzędzi ułatwiających programowanie w modelu rozproszonym.

1.1. Cel i założenia pracy

Celem poniższej pracy jest implementacja w języku Haskell podstawowej biblioteki do rozpraszania zadań na wielu komputerach z wykorzystaniem brokera RabbitMQ, umożliwiającej zlecanie zadań do wykonania, przerywanie zadań będących w trakcie wykonywania, raportowanie na bieżąco postępów wykonania, przesyłania wyników oraz wprowadzania zależności pomiędzy zadaniami.

1.2. Zawartość pracy

W rozdziale 2 zostały opisane istniejące rozwiązania, zaimplementowane w innych językach programowania.

Rozdział 3 opisuje najważniejsze aspekty implementacyjne biblioteki rozpraszającej zadania w języku Haskell.

Podsumowanie zawarte w rozdziale 4 zawiera ogólne wnioski, zakres zrealizowanych celów pracy oraz opis funkcjonalności, o które można rozbudować w przyszłości zaimplementowaną wcześniej bibliotekę.

2. Istniejące rozwiązania

Poniższa lista zawiera skrócony opis niektórych bibliotek programistycznych, służących do obliczeń rozproszonych:

2.1. Celery

Celery jest asynchroniczną kolejką zadań opartą o rozproszone komunikaty przesyłane między komputerami, napisaną w języku Python. Działa w czasie rzeczywistym, jednak umożliwia również szeregowanie zadań. Interfejs programistyczny pozwala na zlecanie zadań zarówno w sposób synchroniczny jak i asynchroniczny, oraz przesyłanie wyników. Wspiera wiele brokerów wiadomości (np. RabbitMQ, Redis, MongoDB).

Architektura składa się z biblioteki klienckiej oraz modułu uruchamiającego zadania. Kod zadań jest definiowany w oddzielnym module. Zadania są uruchamiane wsadowo (nie jest możliwa komunikacja z zadaniem po jego uruchomieniu).

Rozbudowane API pozwala na kompozycję wykonywanych zadań, tworzenie złożonych topologii, uruchamianie zadań okresowych oraz zaawansowane monitorowanie oraz zarządzanie zadaniami. Monitorowanie postępów jest możliwe za pomocą rozszerzeń.

2.2. Resque

Resque jest biblioteką języka Ruby, wykorzystującą bazę Redis w charakterze brokera wiadomości. Umożliwia asynchroniczne zlecanie powtarzalnych zadań na innych komputerach. Biblioteka ta jest często używana przez programistów serwisów internetowych do wykonywania długotrwałych operacji (np. generowanie miniaturek zdjęć, rozsyłanie newslettera e-mail, tworzenie raportów, etc...)

Ta biblioteka nie posiada rozbudowanego API i obsługuje jedynie najprostszą możliwość uruchamiania kodu na innym komputerze. Posiada jednak wbudowany panel webowy, umożliwiający proste nadzorowanie zadań w toku oraz węzłów połączonych z brokerem.

2.3. Cloud Haskell

Cloud Haskell to biblioteka języka Haskell, udostępniająca warstwę transportową do komunikacji między węzłami (a więc nie wymaga brokera), mechanizm serializacji domknięć pozwalający na zdalne uruchamianie funkcji zdefiniowanych w kodzie, oraz API do programowania rozproszonego wymodelowane w sposób zbliżony do OTP [2.4]. Biblioteka nie jest skupiona wokół koncepcji zadań, przez co jest zdecydowanie bardziej ogólna od pozostałych rozwiązań.

2.4. Open Telecom Platform

OTP to kolekcja narzędzi zawierających maszynę wirtualną języka Erlang (uruchamianą na wszystkich węzłach), protokół komunikacyjny, brokera CORBA[1], narzędzia do statycznej analizy rozproszonego kodu, rozproszony serwer bazodanowy oraz wiele innych bibliotek. Umożliwia zaimplementowanie zadań, wraz ze wszystkimi dodatkowymi funkcjonalnościami jednak jest znacznie bardziej ogólna i sama nie dostarcza abstrakcji służących do tego celu.

[1] Common Object Request Broker Architecture - standard komunikacyjny, umożliwiający rozproszony dostęp do obiektów znajdujących się na innych węzłach

3. Implementacja rozpraszania zadań

Poniższy rozdział został poświęcony szczególnie istotnym aspektom implementacyjnym biblioteki umożliwiającej uruchamianie zadań na wielu różnych komputerach w języku Haskell, ze szczególnym uwzględnieniem dodatkowych bibliotek i rozszerzeń tego języka.

3.1. Broker RabbitMQ

RabbitMQ to otwartoźródłowy broker wiadomości - oprogramowanie zapewniające odporny na zakłócenia mechanizm komunikacji sieciowej. Został zaimplementowany w języku Erlang z wykorzystaniem Open Telecom Platform. Posiada biblioteki dla wszystkich znaczących języków programowania - w tym dla Haskella (biblioteka amqp - od ang. Advanced Message Queuing Protocol). Dwie najważniejsze koncepcje implementowane przez RabbitMQ to - **kolejki** (ang. queues) oraz **wymienniki** (ang. exchanges). Kolejki to przetrzymywane przez serwer struktury FIFO z sieciowym protokołem dostępu. Są najbardziej podstawową abstrakcją stosowaną w rozproszonym programowaniu współbieżnym, a w przypadku projektu będącego przedmiotem tej pracy pozwalają wielu węzłom ”konkurować” o zadania w bezpieczny sposób (nigdy nie dojdzie do sytuacji, w której dwa zadania są w tym samym czasie wykonywane na dwóch różnych węzłach). Wymienniki są pośrednikami dostępu do zbiorów kolejek - wiadomość, która zostanie dodana do wymiennika może zostać powielona i rozdystrybuowana do połączonych z wymiennikiem kolejek w jeden z czterech możliwych sposobów zależnie od predefiniowanego typu wymiennika:

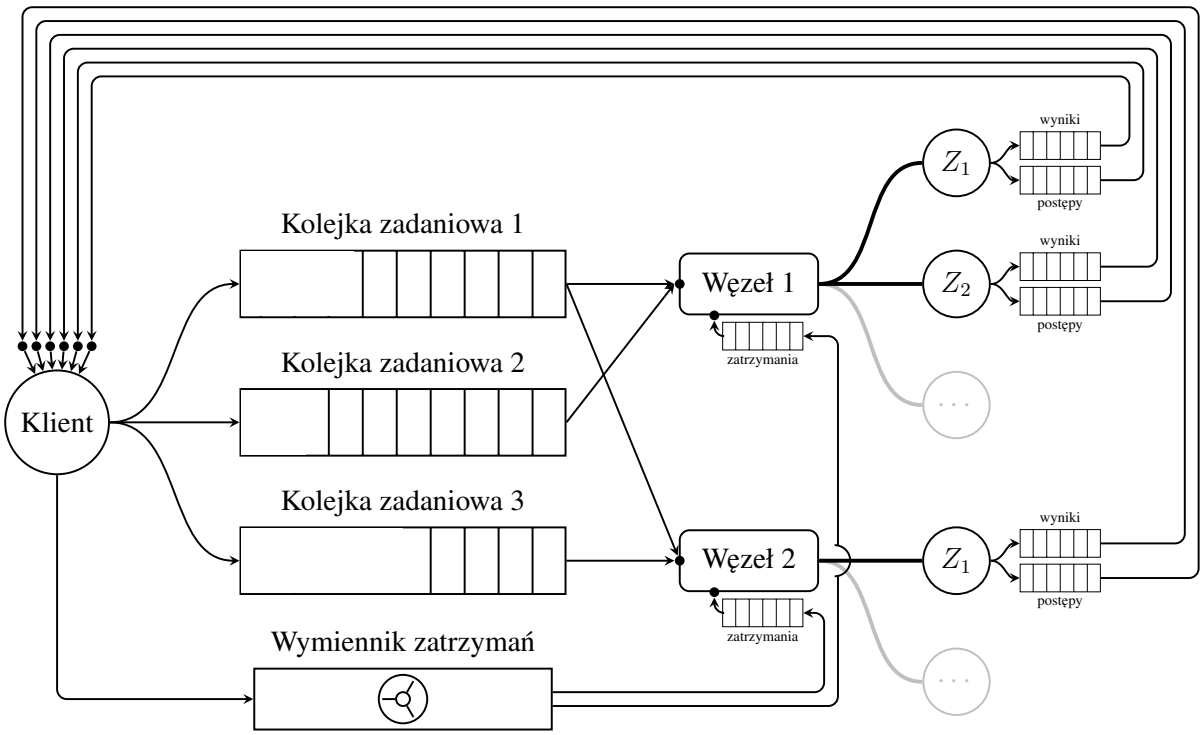
direct – wiadomość trafia tylko do kolejek z określonym kluczem routującym, identycznym z kluczem zawartym w nagłówku wiadomości

fanout – wiadomość trafia do wszystkich podłączonych kolejek

topic – wiadomość trafia do konkretnej kolejki tylko wtedy, kiedy jej klucz routujący spełnia wzorzec określony dla tej kolejki

headers – wiadomość trafia do konkretnej kolejki na podstawie innych niż klucz routujący parametrów zawartych w nagłówku

Topologia wykorzystywana w projekcie zakłada, że każdy węzeł może nasłuchiwać więcej niż jednej kolejki zadaniowej (jest to pożądane w przypadku kiedy nie każdy węzeł jest w stanie obsłużyć każdy typ zadania). Dodatkowo każdy węzeł tworzy własną kolejkę na zlecenia przerwania zadania i przypina ją do wspólnego wymiennika typu `fanout`. Gwarantuje to dostarczenie każdemu węzłowi komunikatu przerwania niezależnie od awarii komunikacji z brokerem. Topologia kolejkowania została przedstawiona na rysunku 3.1.



Rys. 3.1. Topologia kolejkowania

3.2. Zarządzanie zasobami

Protokół AMQP operuje na zależnych od siebie zasobach, którymi należy w poprawny sposób zarządzać. Postawowy przykład z dokumentacji biblioteki AMQP (Listing 3.1), w przypadku wystąpienia wyjątku nie gwarantuje zwolnienia zasobu, a kolejność wykonywania finalizatorów ze względu na leniwą ewaluację jest determinowana wyłącznie wyzwoleniem mechanizmu odśmiecania.

```
1 main = do
2   conn <- openConnection "127.0.0.1" "/" "guest" "guest"
3   ...
4   closeConnection conn
```

Listing 3.1. Łączenie z RabbitMQ

Jednym z istniejących rozwiązań tego problemu jest biblioteka `io-region` [1], umożliwiająca podział kodu na regiony odpowiedzialne za poszczególne zasoby (Listing 3.2) oraz przenoszenie tych odpowiedzialności pomiędzy regionami.

```
...
region $ \r -> do
 connection <- alloc_ r (AMQP.openConnection host vhost username password)
                         AMQP.closeConnection
  -- po opuszczeniu regionu następuje zwolnienie zasobów
...
```

Listing 3.2. Regionalizacja zasobów

Rejestrowanie zasobów w obrębie odpowiednich regionów rozwiązuje również problem asynchronicznych wywołań zwrotnych, służących do obsługi nadchodzących komunikatów z brokera. Dla porównania, niewłaściwe rozwiązanie oparte o mechanizm **`bracket`** (przedstawione na listingu 3.3) powoduje przedwczesne zamknięcie zasobu nadal wykorzystywanego przez funkcję uruchamianą asynchronicznie.

```
...
bracket openConnection closeConnection $ \connection ->
  bracket (openChannel connection) closeChannel $ \channel1 ->
    consumeMsgs channel1 queue callback -- wywołanie asynchroniczne
  -- nieporządane zamknięcie kanału
  -- otwarcie drugiego kanału
  bracket (openChannel connection) closeChannel $ \channel2 ->
    consumeMsgs channel2 queue callback -- wywołanie asynchroniczne
    -- nieporządane zamknięcie kanału
  _ <- getLine -- oczekiwanie
-- zamknięcie połączenia

...
region $ \r -> do
  connection <- alloc_ r openConnection closeConnection
  channel1 <- alloc_ r (openChannel connection) closeChannel
  consumeMsgs channel1 queue callback
  channel2 <- alloc_ r (openChannel connection) closeChannel
  onsumeMsgs channel2 queue callback

  _ <- getLine -- oczekiwanie
  -- zwolnienie zasobów w poprawnej kolejności
```

Listing 3.3. Problem funkcji asynchronicznych

3.3. Środowisko funkcji

Większość funkcji związanych z protokołem AMQP wymaga do działania przekazania pewnego zasobu (jak na przykład obiekt `Connection` lub `Channel`). Robienie tego za każdym razem explicite prowadzi do zmniejszenia czytelności kodu. Idiomatycznym dla języka Haskell rozwiązaniem jest zastosowanie przedstawionego na listingu 3.4 typu `Reader` [2].

```
1  newtype Reader e a = Reader { runReader :: e -> a }
2
3  instance Functor (Reader e) where
4    fmap f r = Reader $ \e -> f (runReader r e)
5
6  instance Applicative (Reader e) where
7    pure a    = Reader $ \e -> a
8    ra <*> rb = Reader $ \e -> (runReader ra e) (runReader rb e)
9
10 instance Monad (Reader e) where
11   (Reader r) >>= f = Reader $ \e -> runReader (f (r e)) e
12
13 ask :: Reader a a
14 ask = Reader id
```

Listing 3.4. Typ reader

Typ `Reader` opakowuje funkcję przyjmującą jako argument środowisko jej wykonania i zwracającej pewien wynik obliczeń wykorzystujących to środowisko. Przykładowo dla:

```
1  testReader :: Reader Bool String
2  testReader = Reader $ \flag -> if flag then "Włącz" else "Wyłącz"
```

wywołanie `runReader testReader True` zwróci wartość `"Włącz"` – jednak istotą działania `Reader`'a jest jego monadyczny interfejs umożliwiający zapisanie funkcji `testReader` jako:

```
1  testReader :: Reader Bool String
2  testReader = do
3    flag <- ask
4    return $ if flag then "Włącz" else "Wyłącz"
```

Dzięki temu unikamy przekazywania tego samego środowiska za każdym razem jako argumentu funkcji, co jest szczególnie istotne w przypadku kiedy jest to wiele funkcji.

Niestety funkcjonalność samej monady `Reader` nie jest wystarczająca ze względu na mnogość operacji wykorzystujących operacje wejścia-wyjścia, więc wymagających typu `IO` do działania. Pomocne okazują się przedstawione na listingu 3.5 transformatory monad (ang. *Monad transformers* [3]):

```haskell
newtype ReaderT e m a = ReaderT { runReaderT :: e -> m a }

instance Functor m => Functor (ReaderT e m) where
  fmap f r = ReaderT $ \e -> fmap f (runReaderT r e)

instance Applicative m => Applicative (ReaderT e m) where
  pure a    = ReaderT $ \e -> pure a
  ra <*> rb = ReaderT $ \e -> (runReaderT ra e) <*> (runReaderT rb e)

instance Monad m => Monad (ReaderT e m) where
  r >>= f = ReaderT $ \e -> do
    a <- runReaderT r e
    runReaderT (f a) e

ask :: Monad m => ReaderT a m a
ask = ReaderT return

lift :: m a -> ReaderT e m a
lift m = ReaderT (const m)
```

Listing 3.5. Transformator typu Reader

Dzięki funkcji `lift` możemy niejako „podciągać” operacje wykonane w ramach innej monady do typu `ReaderT`:

```haskell
testReader :: ReaderT Bool IO String
testReader = do
  flag <- ask
  lift $ if flag then putStrLn "Włącz"
                 else putStrLn "Wyłącz"
  return "Wynik"

> runReaderT testReader True
Włącz
```

3.4. Odczyt konfiguracji z pliku

Biblioteka `configurator` [4] umożliwia odczyt plików konfiguracyjnych i zapewnia uproszczony interfejs dostępu do przechowywanych wewnątrz wartości parametrów, a w przypadku ich braku korzysta z wartości domyślnej zapisanej „na sztywno” w kodze konfigurowanego programu, co dla poniższego (3.6) pliku konfiguracyjnego zostało zaimplementowane na listingu 3.7.

```
taskell {
  rabbitmq {
    host     = "localhost"
    vhost    = "/"
    username = "guest"
    password = "guest"
  }
  abortExchange = "taskell.abort"
  parallelism = 1
  queues = ["taskell.q1", "taskell.q2"]
}
```

Listing 3.6. Przykładowy plik konfiguracyjny

taskell.rabbitmq.host – Nazwa sieciowa lub adres IP brokera

taskell.rabbitmq.vhost – URL hosta wirtualnego

taskell.rabbitmq.username – Nazwa użytkownika skonfigurowana na brokerze

taskell.rabbitmq.password – Hasło użytkownika skonfigurowane na brokerze

taskell.abortExchange – Nazwa wymiennika do którego każdy węzeł przypina swoją kolejkę celem nasłuchu zleceń przerwania zadana

taskell.queues – Lista kolejek, których węzeł ma nasłuchiwać w oczekiwaniu na zadanie

taskell.parallelism – Liczba zadań, które węzeł może przetwarzać jednocześnie

```
main = do
  [cp] <- getArgs
  config <- load [ Required cp ]
  host     <- lookupDefault "localhost" config "taskell.rabbitmq.host"
  vhost    <- lookupDefault "//"        config "taskell.rabbitmq.vhost"
  username <- lookupDefault "guest"     config "taskell.rabbitmq.username"
  password <- lookupDefault "password"  config "taskell.rabbitmq.password"
```

Listing 3.7. Odczyt konfiguracji

3.5. Obsługa zadań

Mechanizm uruchamiania konkretnego zadania sprowadza się do odczytu odpowiedniej funkcji z tablicy mieszającej zawierającej wszystkie obsługiwane przez węzeł zadania na podstawie klucza będącego wybraną przez programistę nazwą zadania, a następnie uruchomienie (po odpowiednim sparametryzowaniu) tej funkcji wewnątrz oddzielnego wątku. Ogólny schemat przedstawia listing 3.8.

```
1  newtype Task = Task { runTask :: ByteString -> IO ByteString }
2
3  registeredTasks :: HashMap Text Task
4  registeredTasks = fromList [ ("name1", function1)
5                             , ("name2", function2)
6                             , ("name3", function3) ]
7  ...
8
9  handleTask (msg, env) = do
10   let Just taskName = AMQP.msgType msg
11   let      taskArgs = AMQP.msgBody msg
12
13   forkIO $
14     result <- runTask (registeredTasks ! taskName) taskArgs
15     ...
16
17   AMQP.ackEnv env
```

Listing 3.8. Schemat obsługi zadania

Nieprzypadkowo typ `Task` to opakowana, monomorficzna funkcja przetwarzająca ciągi bajtów. Gdyby pokusić się o przeniesienie odpowiedzialności za deserializację argumentów i serializację wyników obliczeń, typ ten przyjąłby pozornie „bezpieczniejszą” formę egzystencjalną:

$$\forall_{a,b}(\text{Serializable } a, \text{Serializable } b) \Rightarrow a \rightarrow b$$

jednak w takiej sytuacji kompilator nie może ustalić typów polimorficznych a, b w kontekście fragmentu kodu obliczającego wartość, co zostało zobrazowane na listingu 3.9

```haskell
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE RankNTypes                #-}
module Test where

import Data.Store
import Data.Text
import Data.ByteString
import Data.HashMap.Strict

newtype Task = Task { runTask :: forall a b . (Store a, Store b)
                              => a -> IO b }

registeredTasks :: HashMap Text Task
registeredTasks = ...

runTaskByName :: Text -> ByteString -> IO ByteString
runTaskByName taskName encodedArg = do
  arg <- decodeIO encodedArg
  result <- runTask (registeredTasks ! taskName) arg
  -- Błąd sprawdzania jednoznaczności typów
  return $ encode result
```

Listing 3.9. Problem typu zadania zdefiniowanego egzystencjalnie

Błąd ten jest analogiczny do przedstawionego na listingu 3.10 bardziej podstawowego przykładu i wychwytuje go mechanizm sprawdzania jednoznaczności instancjonowanych typów.

```haskell
show (read "5" :: Int)          => "5"
show (read "5" :: Double)       => "5.0"
show (read "5") => Ambiguous type variable ‘a2’ arising from a use of ‘read’
                   prevents the constraint ‘(Read a2)’ from being solved.
```

Listing 3.10. Podstawowy przykład niejednoznaczności typów

Inferencja typu dla trzeciego wyrażenia przebiega w systemie Hindleya-Milnera [5] rozszerzonym o klasy typów [6] następująco: Najpierw zostają wprowadzone (reguła `Var`) potrzebne wartości. Po zastosowaniu reguły eliminacji aplikacji funkcji (`App`) powstaje kwantyfikowany egzystencjalnie element ograniczony klasą `Read`, który można zaaplikować (z użyciem reguły eliminacji aplikacji funkcji z łączeniem kontekstów `Comb`) jako argument funkcji `show`, co prowadzi do wynikowego typu `String`, jednak kwantyfikowanego kontekstem odnoszącym się do zmiennej α niewystępującej poza nim, co nie jest dozwolone zgodnie z regułą jednoznaczności typów opisaną w [7]). Diagram inferencji przedstawia rysunek 3.2

$$\dfrac{\dfrac{\text{show} : \forall\alpha.\ (\text{Show}\ \alpha).\ \alpha \to \text{String} \in \Gamma}{\Gamma \vdash \text{show} : \forall\alpha.\ (\text{Show}\ \alpha).\ \alpha \to \text{String}}\ [\text{Var}] \qquad \dfrac{\dfrac{\langle 5\rangle : \text{String} \in \Gamma}{\Gamma \vdash \langle 5\rangle : \text{String}}\ [\text{Var}] \qquad \dfrac{\text{read} : \forall\alpha.\ (\text{Read}\ \alpha).\ \text{String} \to \alpha \in \Gamma}{\Gamma \vdash \text{read} : \forall\alpha.\ (\text{Read}\ \alpha).\ \text{String} \to \alpha}\ [\text{Var}]}{\text{read}\ \langle 5\rangle : \forall\alpha.\ (\text{Read}\ \alpha).\alpha}\ [\text{App}]}{\text{show}\big(\text{read}\ \langle 5\rangle\big) : \forall\alpha.\ (\text{Read}\ \alpha, \text{Show}\ \alpha).\ \text{String}}\ [\text{Comb}]$$

Rys. 3.2. Inferencja niejednoznacznego typu

3.6. Raportowanie postępów wykonania

Każde zadanie oprócz zwracania rezultatu może również raportować na bieżąco postęp wykonywanych operacji. Skuteczną metodą obsługi tego mechanizmu są współprogramy (ang. *coroutines*). Biblioteka `monad-coroutine` [8] dostarcza spójny i idiomatyczny interfejs pozwalający na zaimplementowanie tej funkcjonalności, co przedstawia listing 3.11.

```
1  newtype Task arg p r = Task { runTask :: arg -> Coroutine (Yield p) IO r }
2
3  instance Functor (Task arg p) where
4    fmap f (Task t) = Task $ \arg -> fmap f (t arg)
5
6  instance Applicative (Task arg p) where
7    pure t = Task $ \_ -> pure t
8    Task t1 <*> Task t2 = Task $ \arg -> t1 arg <*> t2 arg
9
10 instance Monad (Task arg p) where
11   (Task t) >>= f = Task $ \arg -> do
12     t' <- t arg
13     runTask (f t') arg
14
15 type TaskStore = HashMap Text RawTask
16
17 runTaskByName :: MonadIO m => TaskStore -> Text
18                -> ByteString           -- Zserializowany argument zadania
19                -> (ByteString -> IO a) -- Domknięcie raportujące postępy
20                -> m ByteString         -- Wynik zadania
21 runTaskByName ts key arg reportFn = liftIO $ do
22   let producer = runTask (ts ! key) arg
23   pogoStick (\(Yield x cont) -> lift (reportFn x) >> cont) producer
24
25 progress :: Monad m => p -> Coroutine (Yield p) m ()
26 progress = yield
```

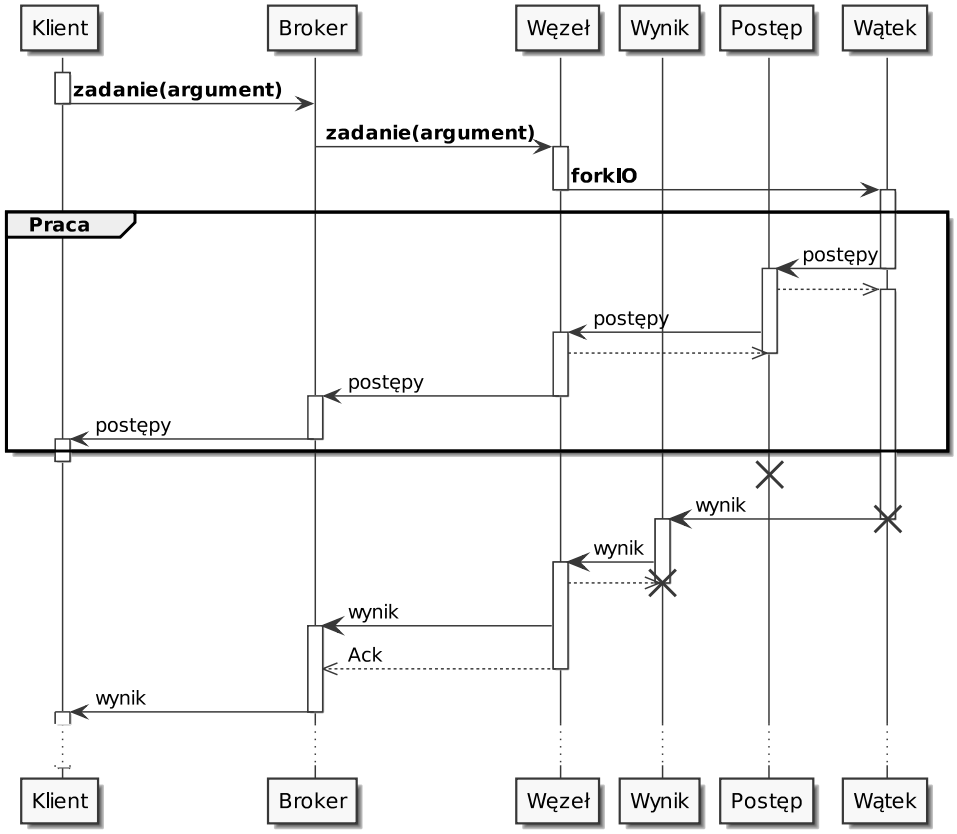
Listing 3.11. Ostateczna implementacja zadania

Przykładowe zadanie sumujące liczby i raportujące postęp po przetworzeniu każdego elementu listy wygląda wtedy następująco:

```
summationTask :: Task [Int] Int Int
summationTask = Task $ \numbers ->
  let indexedNumbers = zip numbers [1..]
      count = fromIntegral $ length numbers
      sumf acc (a, nr) = do
      progress $ floor $ (fromIntegral nr / count) * 100
      return $ acc + a
  in foldM sumf 0 indexedNumbers
```

3.7. Schemat komunikacji z brokerem

Zadania są definiowane przez użytkownika, a więc rozsądnym wydaje się odizolowanie wykonywanego kodu w oddzielnym wątku, tak aby ograniczyć opóźnienia komunikacyjne. Poniższy diagram sekwencji obrazuje proces zachodzący podczas wykonywania przykładowego zadania, przy założeniu, że całość komunikacji odbywa się poprzez transakcyjne zmienne synchronizacyjne [9] co zostało przedstawione na diagramie 3.3.



Rys. 3.3. Diagram sekwencji wykonywania zadania

Kiedy postępy są zapisywane w zmiennej transakcyjnej, wykonanie wątku jest przerywane tylko na czas tego konkretnego zapisu, a przesył postępów odbywa się w innym wątku. Takie podejście minimalizuje przestoje, przy założeniu, że wysyłanie oczekujących komunikatów odbywa się szybciej niż produkcja nowych. W przeciwnym wypadku następuje blokowanie przy próbie zapisu do zmiennej, z której inny wątek nie zdążył jeszcze pobrać wartości. Obsługa przesyłania rezultatu zadania jest identyczna z obsługą postępów, co w przyszłości umożliwia zaimplementowanie zadań produkujących więcej niż jeden wynik.

Zatwierdzenie odebrania zadania odbywa się dopiero po odesłaniu wyniku, co pozwala na wykorzystanie wbudowanego w RabbitMQ mechanizmu automatycznego ponownego kolejkowania niedokończonych zadań, na wypadek np. fizycznej awarii węzła.

3.8. Przerywanie zadań

Przerywanie zadań jest obsługiwane poprzez zapamiętanie identyfikatora wątku zadania w transakcyjnej mapie [10], gdzie kluczami są identyfikatory aktualnie wykonywanych zadań, a wartościami identyfikatory wątku obsługującego zadanie. Komunikat przerwania zadania zawiera identyfikator zadania, które ma zostać przerwane, co umożliwia przerwanie odpowiedniego wątku. Przykładową implementację zawiera listing 3.12

```
1  type CurrentTasks = STM.Map UUID ThreadId
2
3  abortHandler :: CurrentTasks -> ReaderT (Env (Message, Envelope)) IO ()
4  abortHandler currentTasks = do
5    Env r log (msg, env) <- ask
6    deferAck r env
7
8    let Just taskId = fromASCIIBytes $ BL.toStrict (msgBody msg)
9    log $ "Got abort for task " <> toText taskId
10
11   let strategy k = return (k, STM.Focus.Remove)
12   abortCurrent <- atomically $ STM.focus strategy taskId currentTasks
13   case abortCurrent of
14     Just threadId -> do
15         liftIO $ killThread threadId
16         log "Abort received, thread killed"
17     Nothing -> log "Not my task, skipping"
```

Listing 3.12. Przerywanie zadań

3.9. Klient

Uruchamianie zadań jest możliwe za pomocą funkcji `enqueueTask`, a oczekiwanie na wyniki i postępy umożliwiają odpowiednio funkcje `onResult` oraz `onProgress` zdefiniowane następująco:

```
enqueueTask :: (MonadIO m, MonadReader (Env Channel) m)
            => T.Text -> T.Text -> BL.ByteString -> m UUID
enqueueTask qname taskName taskArgs = do
  taskId <- liftIO nextRandom
  queue newQueue { queueName = qname, queueDurable = False
                 , queueExclusive = True} $ do
    publish newMsg { msgBody = taskArgs
                   , msgID = Just $ toText taskId
                   , msgType = Just taskName
                   , msgDeliveryMode = Just Persistent }
  return taskId

onSuffix :: (MonadIO m, MonadReader (Env Channel) m)
         => T.Text -> UUID -> (BL.ByteString -> ReaderT (Env Channel) IO ())
         -> m ()
onSuffix suffix taskId callback = do
  env <- ask
  queue newQueue { queueName = toText taskId <> suffix
                 , queueAutoDelete = True} $
    subscribe Ack $ do
      Env r _ (msg, envelope) <- ask
      deferAck r envelope
      liftIO $ runReaderT (callback $ msgBody msg) env
  return ()

onProgress = onSuffix ".progress"
onResult = onSuffix ".result"
```

Listing 3.13. Klient uruchamiający zadania

Uruchomienie zadania polega na umieszczeniu na wybranej przez użytkownika kolejce komunikatu zlecającego zadanie, którego nasłuchują węzły. Kiedy węzeł odeśle postęp lub rezultat, możliwe jest zlecenie kolejnego zadania, w szczególności wykorzystującego rezultat poprzedniego jako swój argument. Dzięki niskopoziomowemu interfejsowi możemy wykorzystywać wiele różnych strategii serializacji i deserializacji przesyłanych danych oraz eliminować ich izomorficzne przekształcenia (niepotrzebne transformacje danych będących w odpowiednich formatach). Kosztem takiego rozwiązania jest brak bezpieczeństwa typów.

Przykład uruchamiający zadanie, wykorzystujący do serializacji bibliotekę `store` został zawarty na listingu 3.14

```haskell
{-# LANGUAGE ViewPatterns #-}

deserialize :: Store a => BL.ByteString -> a
deserialize = decodeEx . BL.toStrict

serialize :: Store a => a -> BL.ByteString
serialize = BL.fromStrict . encode

printL :: (MonadIO m, Show a) => a -> m ()
printL = liftIO . print

main :: IO ()
main = logger defaultLoger $ connection "localhost" "/" "guest" "guest" $ do
  channel "task" $ do
    task1 <- enqueueTask "taskell.q1" "additionTask"
                              $ serialize (1 :: Int, 2 :: Int)
    task1 `onProgress` \(deserialize -> p) -> printL (p :: Int)
    task1 `onResult` \r -> do
      task2 <-  enqueueTask "taskell.q1" "dummyTask" r
      task2 `onResult` \(deserialize -> p) -> printL (p :: Int)
```

Listing 3.14. Uruchamianie zadania

4. Podsumowanie

W ramach niniejszej pracy została zrealizowana praktyczna implementacja biblioteki umożliwiającej rozpraszanie zadań na wielu komputerach (Rozdział 3), ich nadzorowanie, przerywanie (Sekcja 3.8), a także raportowanie w czasie rzeczywistym postępów (Sekcja 3.6). W ten sposób zostały osiągnięte główne cele pracy.

Rozpraszanie zadań jest niezmiernie złożonym zagadnieniem, a jego odpowiednie ujęcie w kontekście architektury oprogramowania to wyzwanie, z którym każdy programista mierzy się niejednokrotnie w swojej karierze. Języki funkcyjne pozwalają na opisywanie tych zagadnień za pomocą różnorodnych abstrakcji matematycznych, dających nieznane dotąd możliwości analizy struktury programów rozproszonych.

4.1. Możliwe dalsze kierunki rozwoju

Zakres tej pracy pozwolił na zaimplementowanie funkcjonalnej, jednak ograniczonej z punktu widzenia użytkownika końcowego biblioteki. Podczas jej powstawiania zostały zidentyfikowane następujące obszary, które można usprawnić:

Konfiguracja

W obecnej implementacji każdorazowa edycja pliku konfiguracyjnego wymaga restartowania całego modułu obsługi zadań na węźle. Biblioteka `configurator` posiada wbudowany mechanizm powiadamiający o zmianach, jednak nie jest on obecnie wykorzystywany.

Ponadto w obecnej wersji nie jest przeprowadzana weryfikacja poprawności konfiguracji, co przekłada się na trudniejsze wychwytywanie nawet najprostszych błędów.

Obsługa błędów

Jednym z większych wyzwań architektonicznych jest obsługa błędnych ścieżek wykonania oraz sytuacji nadzwyczajnych (wyjątków). W wielowątkowym, rozproszonym środowisku jest to szczególnie trudne, ponieważ błędy mogą występować w bardzo wielu, często asynchronicznych scenariuszach. Dodatkowo język Haskell nie posiada pojedynczego, ustandaryzowanego mechanizmu ich obsługi co czyni poprawną implementację szczególnie trudną.

Bezpieczeństwo typów

Kiedy język Haskell zyska pełny system typów zależnych, będzie możliwe zapisanie typu zadania w sposób umożliwiający automatyczną inferencję sposobu serializacji danych, oraz wychwytujący błędy typów w kodzie klienta (jednak w dalszym ciągu nie będzie możliwe sprawdzanie spójności typów między kodem modułu zadań a klientem, ponieważ są to moduły zupełnie od siebie niezależne).

Obsługa zadań interaktywnych

Obecnie kod zadania jest uruchamiany za każdym razem, kiedy zadanie zostaje zlecone, oraz odpowiada pojedynczym wynikiem po zakończeniu pracy. Przy wykorzystaniu współprogramów jest jednak możliwa implementacja zadań, które po uruchomieniu oczekują serii argumentów i dla każdego kolejnego argumentu produkują nowy wynik, nie kończąc swojej pracy.

Serializacja domknięć oraz dynamiczne ładowanie kodu

Aby uruchomić kod zadania, w obecnej wersji musi ono zostać umieszczone w odpowiednim module, statycznie konsolidowanym wraz z modułem węzła. Takie podejście gwarantuje bezpieczeństwo binarnej kompatybilności, jednak powoduje znaczną komplikację wdrażania poprawek w przypadku architektury składającej się z wielu węzłów.

Za pomocą mechanizmu serializacji domknięć możliwe byłoby zdalne uruchamianie funkcji zdefiniowanych po stronie klienta. Projekt `Cloud Haskell` posiada mechanizm serializacji domknięć, jednak jego implementacja jest jeszcze w fazie rozwojowej.

Mechanizm dynamicznego ładowania kodu umożliwiałby łatwą aktualizację oprogramowania węzłów i może zostać zaimplementowany z wykorzystaniem API trybu interaktywnego `GHCi`.

Topologia zadań na poziomie brokera

Istniejąca implementacja pozwala na wprowadzanie zależności między zadaniami, jednak za każdym razem wyniki i argumenty są przesyłane do klienta, pełniącego centralną rolę. Wspomniany wcześniej mechanizm zadań interaktywnych, po odpowiednim rozszerzeniu pozwalałby zadaniu pobierać argumenty bezpośrednio z kolejki, do której produkuje wyniki inne zadanie.

Inne funkcjonalności

Biblioteki do rozpraszania zadań posiadają bardzo wiele unikalnych rozwiązań, które z powodzeniem można przenieść na grunt języków funkcyjnych i rozszerzyć wykorzystując abstrakcje i idiomy, istniejące jedynie w tych językach - pozwalając na jeszcze wydajniejsze i bardziej przyjazne dla programistów tworzenie oprogramowania rozproszonego.

Bibliografia

[1] Yuras Shumovich. *The io-region package*. 2015. URL: *https://hackage.haskell.org/package/io-region*.

[2] Mark P. Jones. „Functional Programming with Overloading and Higher-Order Polymorphism". W: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995, s. 97–136. ISBN: 3-540-59451-5.

[3] Sheng Liang, Paul Hudak i Mark Jones. „Monad Transformers and Modular Interpreters". W: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, s. 333–343. ISBN: 0-89791-692-1. DOI: *10.1145/199448.199528*.

[4] Bryan O'Sullivan. *The configurator package*. 2014. URL: *https://hackage.haskell.org/package/configurator*.

[5] Luis Damas i Robin Milner. „Principal Type-schemes for Functional Programs". W: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, s. 207–212. ISBN: 0-89791-065-6. DOI: *10.1145/582153.582176*.

[6] P. Wadler i S. Blott. „How to Make Ad-hoc Polymorphism Less Ad Hoc". W: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, s. 60–76. ISBN: 0-89791-294-2. DOI: *10.1145/75277.75283*.

[7] Simon Marlow. „Haskell 2010 Language Report". W: (2010).

[8] Mario Blazevic. *The monad-coroutine package*. 2016. URL: *https://hackage.haskell.org/package/monad-coroutine*.

[9] Tim Harris, Simon Marlow i Simon Peyton Jones. „Composable memory transactions". W: *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, sty. 2005, s. 48–60. ISBN: 1-59593-080-9.

[10] Nikita Volkov. *The stm-containers package*. 2014. URL: *https://hackage.haskell.org/package/stm-containers*.