# An Applicative Application

# Applicative

How to obtain OpenTSDB query batching by exploiting the applicative structure of a computation.

# A Word of Warning

# A Word of Warning

I had trouble explaining this stuff

**Rob Norris** @tpolecat — Oct 04 2017 02:24

Sorry, I need to focus on something else right now. I'm sure someone can help you work through it.

**Ionuț G. Stan** @igstan — Oct 04 2017 02:25

No worries, I like banging my head against this wall actually :)

**Ionuț G. Stan** @igstan — Oct 04 2017 03:06

Well, it doesn't have to return, it's an applicative, so its continuation can be saved somewhere, right?

**Fabio Labella** @SystemFw — Oct 04 2017 03:06

I can't parse that

**Fabio Labella** @SystemFw — Oct 04 2017 13:46

I'm busy today

**Ionuț G. Stan** @igstan — Oct 04 2017 13:46

No worries, you've helped a lot anyways.

**Fabio Labella** @SystemFw — Oct 04 2017 13:46

I'll try to show actual code tonight or tomorrow (UK time)

# Today's Roadmap

# Today's Roadmap

- The Problem

# Today's Roadmap

- The Problem
  - OpenTSDB Overview — Time Series Data Storage

# Today's Roadmap

- The Problem
  - OpenTSDB Overview — Time Series Data Storage
  - Business Use Case — Time Series Data Aggregation

# Today's Roadmap

- The Problem
  - OpenTSDB Overview — Time Series Data Storage
  - Business Use Case — Time Series Data Aggregation
- The Solution

# Today's Roadmap

- The Problem
  - OpenTSDB Overview — Time Series Data Storage
  - Business Use Case — Time Series Data Aggregation
- The Solution
  - Inspiration

# Today's Roadmap

- The Problem
  - OpenTSDB Overview — Time Series Data Storage
  - Business Use Case — Time Series Data Aggregation
- The Solution
  - Inspiration
  - Intuitions — Functions, Monads & Applicatives

# Today's Roadmap

- The Problem
  - OpenTSDB Overview — Time Series Data Storage
  - Business Use Case — Time Series Data Aggregation
- The Solution
  - Inspiration
  - Intuitions — Functions, Monads & Applicatives
  - Code

# Today's Roadmap

- The Problem
  - OpenTSDB Overview — Time Series Data Storage
  - Business Use Case — Time Series Data Aggregation
- The Solution
  - Inspiration
  - Intuitions — Functions, Monads & Applicatives
  - Code
- Future Work

# OpenTSDB Overview

Time Series Data Storage

```
http POST 'localhost:4242/api/put' << EOF
[
  {
    "metric": "metric-a",
    "timestamp": 1538092800,
    "value": 10,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  },
  {
    "metric": "metric-b",
    "timestamp": 1538092801,
    "value": 20,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  }
]
EOF
```

```
http POST 'localhost:4242/api/put' << EOF
[
  {
    "metric": "metric-a",
    "timestamp": 1538092800,
    "value": 10,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  },
  {
    "metric": "metric-b",
    "timestamp": 1538092801,
    "value": 20,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  }
]
EOF
```

```
http POST 'localhost:4242/api/put' << EOF
[
  {
    "metric": "metric-a",
    "timestamp": 1538092800,
    "value": 10,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  },
  {
    "metric": "metric-b",
    "timestamp": 1538092801,
    "value": 20,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  }
]
EOF
```

```
http POST 'localhost:4242/api/put' << EOF
[
  {
    "metric": "metric-a",
    "timestamp": 1538092800,
    "value": 10,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  },
  {
    "metric": "metric-b",
    "timestamp": 1538092801,
    "value": 20,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  }
]
EOF
```

```
http POST 'localhost:4242/api/put' << EOF
[
  {
    "metric": "metric-a",
    "timestamp": 1538092800,
    "value": 10,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  },
  {
    "metric": "metric-b",
    "timestamp": 1538092801,
    "value": 20,
    "tags": {
      "t1": "v1",
      "t2": "v2"
    }
  }
]
EOF
```

```
http POST 'localhost:4242/api/query' << EOF
{
  "start": "2018/05/01 00:10:00",
  "end": "2018/05/01 00:10:05",
  "timezone": "UTC",
  "queries": [
    {
      "metric": "metric-a",
      "aggregator": "none",
      "tags": {}
    },
    {
      "metric": "metric-b",
      "aggregator": "none",
      "tags": {}
    }
  ]
}
EOF
```

localhost

OPENTSDB

Graph | Stats | Logs | Version

From 2018/05/01-00:09:00 To (now) 2018/05/01-00:21:00 Autoreload WxH: 1420x645 Global annotations

metric-a | metric-b | +

Metric: metric-a

Tags

Rate Rate Ctr Right Axis

Rate Ctr Max:

Rate Ctr Reset: 0

Aggregator: sum

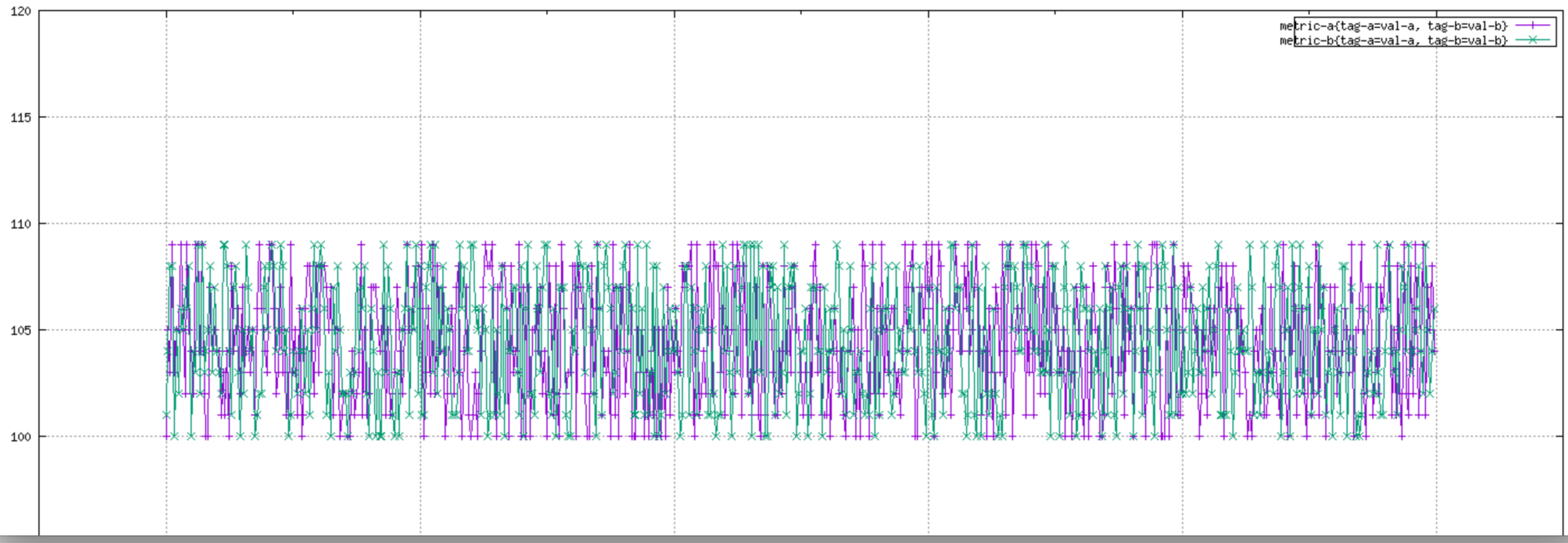
Downsample

avg 10m none

Axes | Key | Style

| | Y | Y2 |
|---|---|---|
| Label | | |
| Format | | |
| Range | [90:12 | |
| Log scale | | |

Cache hit (disk). 1200 points retrieved, 1200 points plotted in 17ms.



# Business Use Case

## Time Series Data Aggregation

# Business Use Case

# Business Use Case

- We store historical data in OpenTSDB

# Business Use Case

- We store historical data in OpenTSDB
- Basis for end-of-month reports

# Business Use Case

- We store historical data in OpenTSDB
- Basis for end-of-month reports
- Past: everything computed at end of month from raw OpenTSDB data

# Business Use Case

- We store historical data in OpenTSDB
- Basis for end-of-month reports
- Past: everything computed at end of month from raw OpenTSDB data
- Now: pre-compute aggregations continuously at various frequencies, e.g., 5 minutes, and use these for reporting

# Business Use Case

- We store historical data in OpenTSDB
- Basis for end-of-month reports
- Past: everything computed at end of month from raw OpenTSDB data
- Now: pre-compute aggregations continuously at various frequencies, e.g., 5 minutes, and use these for reporting
- We call these aggregations roll-ups

# Business Use Case

- We store historical data in OpenTSDB
- Basis for end-of-month reports
- Past: everything computed at end of month from raw OpenTSDB data
- Now: pre-compute aggregations continuously at various frequencies, e.g., 5 minutes, and use these for reporting
- We call these aggregations roll-ups
- Decreases time to generate reports

# Roll-Ups as Functions

```
def rollup(metric: List[Datapoint]): Double
```

```
case class Datapoint(
  timestamp: Instant,
  value: Double,
)

def rollup(metric: List[Datapoint]): Double
```

```
case class Datapoint(
  timestamp: Instant,
  value: Double,
)

def rollup(
  metric0: List[Datapoint],
  metric1: List[Datapoint],
  ...
  metricN: List[Datapoint],
): Double
```

# Goal: Optimize Network Access

# Inspiration

GitHubGist Search... All gists GitHub New gist

chris-taylor / **IOAction.hs**

Last active 2 years ago • Report abuse

★ Star 6 | Fork 1

<> Code | Revisions 2 | ★ Stars 6 | Forks 1 | Embed <script src="https://gist. | Download ZIP

Code for my blog post about pure I/O

IOAction.hs Raw

```haskell
data IOAction a = Return a
                | Put String (IOAction a)
                | Get (String -> IOAction a)

get   = Get Return
put s = Put s (Return ())

seqio :: IOAction a -> (a -> IOAction b) -> IOAction b
seqio (Return a) f = f a
seqio (Put s io) f = Put s (seqio io f)
seqio (Get g)    f = Get (\s -> seqio (g s) f)

echo = get `seqio` put

hello = put "What is your name?"      `seqio` \_    ->
        get                           `seqio` \name ->
        put "What is your age?"       `seqio` \_    ->
        get                           `seqio` \age  ->
        put ("Hello " ++ name ++ "!") `seqio` \_    ->
        put ("You are " ++ age ++ " years old")

hello2 = do put "What is your name?"
            name <- get
            put "What is your age?"
            age <- get
```

**IOAction.hs**

```haskell
data IOAction a = Return a
                | Put String (IOAction a)
                | Get (String -> IOAction a)

get   = Get Return
put s = Put s (Return ())

seqio :: IOAction a -> (a -> IOAction b) -> IOAction b
seqio (Return a) f = f a
seqio (Put s io) f = Put s (seqio io f)
seqio (Get g)    f = Get (\s -> seqio (g s) f)

echo = get `seqio` put

hello = put "What is your name?"      `seqio` \_    ->
        get                           `seqio` \name ->
        put "What is your age?"       `seqio` \_    ->
        get                           `seqio` \age  ->
        put ("Hello " ++ name ++ "!") `seqio` \_    ->
        put ("You are " ++ age ++ " years old")

hello2 = do put "What is your name?"
            name <- get
            put "What is your age?"
            age <- get
```

stackoverflow Search…

# What are free monads?

321

I've seen the term *Free Monad* pop up every now and then for some time, but everyone just seems to use/discuss them without giving an explanation of what they are. So: what are free monads? (I'd say I'm familiar with monads and the Haskell basics, but have only a very rough knowledge of category theory.)

haskell monads free-monad Edit tags

166

share edit close flag

edited May 22 '14 at 8:26

fizruk
1,585 ● 12 ● 23

asked Nov 12 '12 at 21:53

David
3,800 ● 3 ● 19 ● 31

11 A fairly good explanation is here haskellforall.com/2012/06/… – Roger Lindsjö Nov 12 '12 at 22:08

18 @Roger that's kind of the page that brought me here. To me, that example defines a monad instance for a type named "Free" and that's it. – David Nov 12 '12 at 22:12

Apologies for the question but would anyone be able to distill what a free monad is to a c# programmer? I don't understand any of this Haskell mumbo – I--"""---------""""""""" Aug 13 at 11:03

Scala World 2015

20-22 September, Lake District, UK

# Move Over Free Monads: Make Way for Free Applicatives

— John de Goes

Sponsored by

Goldman Sachs Typesafe BOLDRADIUS 47 zalando

0:25 / 45:29

# Free Applicative Functors

Paolo Capriotti
Functional Programming Laboratory
University of Nottingham
pvc@cs.nott.ac.uk

Ambrus Kaposi
Functional Programming Laboratory
University of Nottingham
auk@cs.nott.ac.uk

## Abstract

Applicative functors ([9]) are a generalisation of monads. Both allow expressing effectful computations into an otherwise pure language, like Haskell ([8]).
Applicative functors are to be preferred to monads when the structure of a computation is fixed *a priori*. That makes it possible to perform certain kinds of static analysis on applicative values.
We define a notion of *free applicative functor*, prove that it satisfies the appropriate laws, and that the construction is left adjoint to a suitable forgetful functor.
We show how free applicative functors can be used to implement embedded DSLs which can be statically analysed.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Haskell, Free Applicative Functors

***Keywords*** Applicative Functors, Parametricity, Adjoints

## 1. Introduction

*Free* monads in Haskell are a very well-known and practically used construction. Given any endofunctor `f`, the free monad on `f` is given by a simple inductive definition:

```
data Free f a
  = Return a
  | Free (f (Free f a))
```

The typical use case for this construction is creating embedded

- We give two definitions of *free applicative functor* in Haskell (section 2), and show that they are equivalent (section 5).
- We prove that our definition is correct, in the sense that it really is an applicative functor (section 6), and that it is "free" in a precise sense (section 7).
- We present a number of examples where the use of free applicative functors helps make the code more elegant, removes duplication or enables certain kinds of optimizations which are not possible when using free monads. We describe the differences between expressivity of DSLs using free applicatives and free monads (section 3).
- We compare our definition to other existing implementations of the same idea (section 9).

This paper is aimed at programmers with a working knowledge of Haskell. Familiarity with applicative functors is not required, although it is helpful to understand the motivation behind this work. We make use of category theoretical concepts to justify our definition, but the Haskell code we present can also stand on its own.

### 1.1 Applicative functors

*Applicative functors* (also called *idioms*) were first introduced in [9] as a way to provide a lighter notation for monads. They have since been used in a variety of different applications, including efficient parsing (see section 1.4), regular expressions and bidirectional routing.

# Free Applicative Functors

Paolo Capriotti
Functional Programming Laboratory
University of Nottingham
pvc@cs.nott.ac.uk

Ambrus Kaposi
Functional Programming Laboratory
University of Nottingham
auk@cs.nott.ac.uk

## Abstract

Applicative functors ([9]) are a generalisation of monads. Both allow expressing effectful computations into an otherwise pure language, like Haskell ([8]).
Applicative functors are to be preferred to monads when the structure of a computation is fixed *a priori*. That makes it possible to perform certain kinds of static analysis on applicative values.
We define a notion of *free applicative functor*, prove that it satisfies the appropriate laws, and that the construction is left adjoint to a suitable forgetful functor.
We show how free applicative functors can be used to implement embedded DSLs which can be statically analysed.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Haskell, Free Applicative Functors

***Keywords*** Applicative Functors, Parametricity, Adjoints

## 1. Introduction

*Free* monads in Haskell are a very well-known and practically used construction. Given any endofunctor `f`, the free monad on `f` is given by a simple inductive definition:

```
data Free f a
  = Return a
  | Free (f (Free f a))
```

The typical use case for this construction is creating embedded

- We give two definitions of *free applicative functor* in Haskell (section 2), and show that they are equivalent (section 5).
- We prove that our definition is correct, in the sense that it really is an applicative functor (section 6), and that it is "free" in a precise sense (section 7).
- We present a number of examples where the use of free applicative functors helps make the code more elegant, removes duplication or enables certain kinds of optimizations which are not possible when using free monads. We describe the differences between expressivity of DSLs using free applicatives and free monads (section 3).
- We compare our definition to other existing implementations of the same idea (section 9).

This paper is aimed at programmers with a working knowledge of Haskell. Familiarity with applicative functors is not required, although it is helpful to understand the motivation behind this work. We make use of category theoretical concepts to justify our definition, but the Haskell code we present can also stand on its own.

### 1.1 Applicative functors

*Applicative functors* (also called *idioms*) were first introduced in [9] as a way to provide a lighter notation for monads. They have since been used in a variety of different applications, including efficient parsing (see section 1.4), regular expressions and bidirectional routing.

## Abstract

Applicative functors ([9]) are a generalisation of monads. Both allow expressing effectful computations into an otherwise pure language, like Haskell ([8]).
Applicative functors are to be preferred to monads when the structure of a computation is fixed *a priori*. That makes it possible to perform certain kinds of static analysis on applicative values.
We define a notion of *free applicative functor*, prove that it satisfies the appropriate laws, and that the construction is left adjoint to a suitable forgetful functor.
We show how free applicative functors can be used to implement embedded DSLs which can be statically analysed.



***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Haskell, Free Applicative Functors

***Keywords*** Applicative Functors, Parametricity, Adjoints


## 1. Introduction

*Free* monads in Haskell are a very well-known and practically used construction. Given any endofunctor `f`, the free monad on `f` is given by a simple inductive definition:

```
data Free f a
  = Return a
  | Free (f (Free f a))
```

The typical use case for this construction is creating embedded

- We give two d
  (section 2), and
- We prove that
  is an applicativ
  precise sense (s
- We present a
  plicative functo
  duplication or
  not possible w
  ences between
  free monads (s
- We compare ou
  the same idea (

This paper is aim
of Haskell. Famili
although it is hel
work. We make us
definition, but the
own.

### 1.1 Applicative

*Applicative functor*
as a way to provid
been used in a vari
parsing (see section
ing.

## Abstract

Applicative functors ([9]) are a generalisation of monads. Both allow expressing effectful computations into an otherwise pure language, like Haskell ([8]).

Applicative functors are to be preferred to monads when the structure of a computation is fixed *a priori*. That makes it possible to perform certain kinds of static analysis on applicative values.

We define a notion of *free applicative functor*, prove that it satisfies the appropriate laws, and that the construction is left adjoint to a suitable forgetful functor.

We show how free applicative functors can be used to implement embedded DSLs which can be statically analysed.



***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Haskell, Free Applicative Functors

***Keywords*** Applicative Functors, Parametricity, Adjoints


## 1. Introduction

*Free* monads in Haskell are a very well-known and practically used construction. Given any endofunctor `f`, the free monad on `f` is given by a simple inductive definition:

```
data Free f a
  = Return a
  | Free (f (Free f a))
```

The typical use case for this construction is creating embedded

- We give two d
  (section 2), and
- We prove that
  is an applicativ
  precise sense (s
- We present a
  plicative functo
  duplication or
  not possible w
  ences between
  free monads (s
- We compare ou
  the same idea (

This paper is aim
of Haskell. Famili
although it is help
work. We make us
definition, but the
own.

### 1.1 Applicative

*Applicative functor*
as a way to provide
been used in a vari
parsing (see section
ing.

# Similar Ideas: **HaxI**

# There is no Fork: an Abstraction for Efficient, Concurrent, and Concise Data Access

Simon Marlow
Facebook
smarlow@fb.com

Louis Brandy
Facebook
ldbrandy@fb.com

Jonathan Coens
Facebook
jon.coens@fb.com

Jon Purdy
Facebook
jonp@fb.com

## Abstract

We describe a new programming idiom for concurrency, based on Applicative Functors, where concurrency is implicit in the Applicative `<*>` operator. The result is that concurrent programs can be written in a natural applicative style, and they retain a high degree of clarity and modularity while executing with maximal concurrency. This idiom is particularly useful for programming against external data sources, where the application code is written without the use of explicit concurrency constructs, while the implementation is able to batch together multiple requests for data from the same source, and fetch data from multiple sources concurrently. Our abstraction uses a cache to ensure that multiple requests for the same data return the same result, which frees the programmer from having to arrange to fetch data only once, which in turn leads to greater modularity.

While it is generally applicable, our technique was designed with a particular application in mind: an internal service at Facebook that identifies particular types of content and takes actions based on it. Our application has a large body of business logic that fetches data from several different external sources. The framework described in this paper enables the business logic to execute efficiently by automatically fetching data concurrently; we present some preliminary results.

***Keywords*** Haskell; concurrency; applicative; monad; data-fetching; distributed

## 1. Introduction

concise business logic, uncluttered by performance-related details. In particular the programmer should not need to be concerned with accessing external data efficiently. However, one particular problem often arises that creates a tension between conciseness and efficiency in this setting: accessing multiple remote data sources efficiently requires *concurrency*, and that normally requires the programmer to intervene and program the concurrency explicitly.

When the business logic is only concerned with *reading* data from external sources and not *writing*, the programmer doesn't care about the order in which data accesses happen, since there are no side-effects that could make the result different when the order changes. So in this case the programmer would be entirely happy with not having to specify either ordering or concurrency, and letting the system perform data access in the most efficient way possible. In this paper we present an embedded domain-specific language (EDSL), written in Haskell, that facilitates this style of programming, while automatically extracting and exploiting any concurrency inherent in the program.

Our contributions can be summarised as follows:

- We present an `Applicative` abstraction that allows implicit concurrency to be extracted from computations written with a combination of `Monad` and `Applicative`. This is an extension of the idea of concurrency monads [10], using Applicative `<*>` as a way to introduce concurrency (Section 4). We then develop the idea into an abstraction that supports concurrent access to remote data (Section 5), and failure (Section 8).
- We show how to add a *cache* to the framework (Section 6).

# There is no Fork: an Abstraction for Efficient, Concurrent, and Concise Data Access

Simon Marlow
Facebook
smarlow@fb.com

Louis Brandy
Facebook
ldbrandy@fb.com

Jonathan Coens
Facebook
jon.coens@fb.com

Jon Purdy
Facebook
jonp@fb.com

## Abstract

We describe a new programming idiom for concurrency, based on Applicative Functors, where concurrency is implicit in the Applicative `<*>` operator. The result is that concurrent programs can be written in a natural applicative style, and they retain a high degree of clarity and modularity while executing with maximal concurrency. This idiom is particularly useful for programming against external data sources, where the application code is written without the use of explicit concurrency constructs, while the implementation is able to batch together multiple requests for data from the same source, and fetch data from multiple sources concurrently. Our abstraction uses a cache to ensure that multiple requests for the same data return the same result, which frees the programmer from having to arrange to fetch data only once, which in turn leads to greater modularity.

While it is generally applicable, our technique was designed with a particular application in mind: an internal service at Facebook that identifies particular types of content and takes actions based on it. Our application has a large body of business logic that fetches data from several different external sources. The framework described in this paper enables the business logic to execute efficiently by automatically fetching data concurrently; we present some preliminary results.

***Keywords*** Haskell; concurrency; applicative; monad; data-fetching; distributed

## 1. Introduction

concise business logic, uncluttered by performance-related details. In particular the programmer should not need to be concerned with accessing external data efficiently. However, one particular problem often arises that creates a tension between conciseness and efficiency in this setting: accessing multiple remote data sources efficiently requires *concurrency*, and that normally requires the programmer to intervene and program the concurrency explicitly.

When the business logic is only concerned with *reading* data from external sources and not *writing*, the programmer doesn't care about the order in which data accesses happen, since there are no side-effects that could make the result different when the order changes. So in this case the programmer would be entirely happy with not having to specify either ordering or concurrency, and letting the system perform data access in the most efficient way possible. In this paper we present an embedded domain-specific language (EDSL), written in Haskell, that facilitates this style of programming, while automatically extracting and exploiting any concurrency inherent in the program.

Our contributions can be summarised as follows:

- We present an `Applicative` abstraction that allows implicit concurrency to be extracted from computations written with a combination of `Monad` and `Applicative`. This is an extension of the idea of concurrency monads [10], using Applicative `<*>` as a way to introduce concurrency (Section 4). We then develop the idea into an abstraction that supports concurrent access to remote data (Section 5), and failure (Section 8).
- We show how to add a *cache* to the framework (Section 6).

## Abstract

We describe a new programming idiom for concurrency, based on Applicative Functors, where concurrency is implicit in the Applicative `<*>` operator. The result is that concurrent programs can be written in a natural applicative style, and they retain a high degree of clarity and modularity while executing with maximal concurrency. This idiom is particularly useful for programming against external data sources, where the application code is written without the use of explicit concurrency constructs, while the implementation is able to batch together multiple requests for data from the same source, and fetch data from multiple sources concurrently. Our abstraction uses a cache to ensure that multiple requests for the same data return the same result, which frees the programmer from having to arrange to fetch data only once, which in turn leads to greater modularity.

While it is generally applicable, our technique was designed with a particular application in mind: an internal service at Facebook that identifies particular types of content and takes actions based on it. Our application has a large body of business logic that fetches data from several different external sources. The framework described in this paper enables the business logic to execute efficiently by automatically fetching data concurrently; we present some preliminary results.



***Keywords*** Haskell; concurrency; applicative; monad; data-fetching; distributed


## 1. Introduction

concise business log
In particular the pr
with accessing exte
problem often arises
efficiency in this se
efficiently requires
programmer to inter

When the busine
from external sourc
care about the orde
are no side-effects t
order changes. So i
happy with not havi
and letting the syste
possible. In this pap
language (EDSL), w
programming, while
concurrency inheren

Our contributions

- We present an A
concurrency to b
combination of M
of the idea of co
as a way to introc
the idea into an
remote data (Sec

- We show how t

Applicative lets us perform global optimizations on the Abstract Syntax Tree of an Embedded DSL.

# Intuitions

## Functions, Monads & Applicatives

f(a)

```scala
def example1[A, B](f: A => B, a: A): B =
  f(a)
```

```scala
def example1[A, B](f: A => B, a: A): B =
  f(a)

def example2[F[_]: Applicative, A, B](f: A => B, a: A): F[B] = {
  val ff = Applicative[F].pure(f)
  val fa = Applicative[F].pure(a)
  ff.ap(fa)
}
```

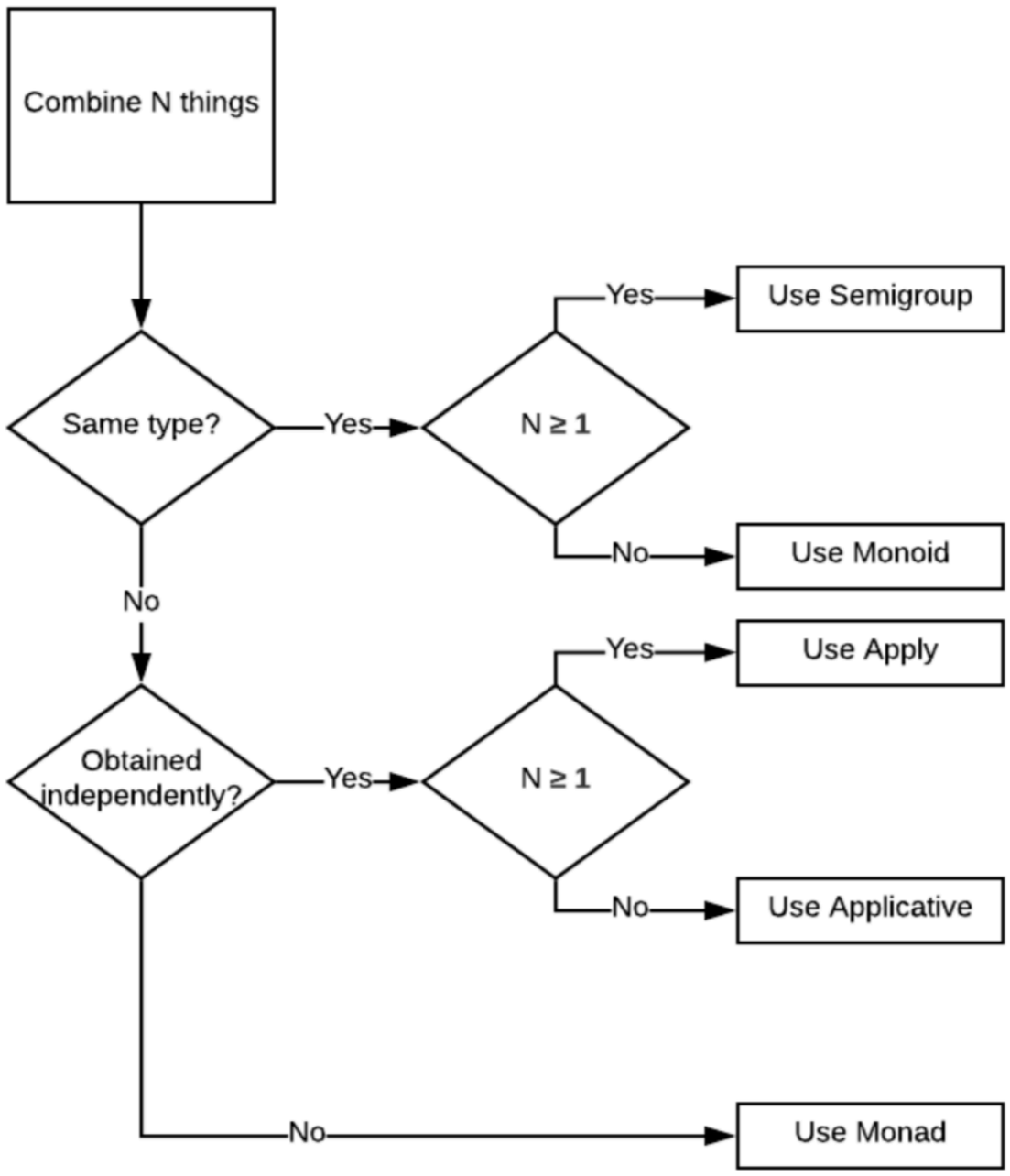
```scala
def example1[A, B](f: A => B, a: A): B =
  f(a)

def example2[F[_]: Applicative, A, B](f: A => B, a: A): F[B] = {
  val ff = Applicative[F].pure(f)
  val fa = Applicative[F].pure(a)
  ff.ap(fa)
}

def example3[F[_]: Applicative, A, B](f: F[A => B], a: F[A]): F[B] =
  f.ap(a)
```

```scala
def example1[A, B](f: A => B, a: A): B =
  f.apply(a)

def example2[F[_]: Applicative, A, B](f: A => B, a: A): F[B] = {
  val ff = Applicative[F].pure(f)
  val fa = Applicative[F].pure(a)
  ff.ap(fa)
}

def example3[F[_]: Applicative, A, B](f: F[A => B], a: F[A]): F[B] =
  f.ap(a)
```

# Applicative

Lets us embed function-like DSLs into our programs.

# Data and Effect Dependencies


Combine N things
Same type?
Yes
N ≥ 1
Yes
Use Semigroup
No
Use Monoid
No
Obtained independently?
Yes
N ≥ 1
Yes
Use Apply
No
Use Applicative
No
Use Monad

# Functions: 1

Possible Side-Effects
Without Data-Dependencies

```
// Potential side-effects, but we don't know.
def foo(): String = ???
def bar(): String = ???
```

```
// Potential side-effects, but we don't know.
def foo(): String = ???
def bar(): String = ???

// No data dependency between function calls.
val a = foo()
val b = bar()
```

# Functions: 2

## Possible Side-Effects
## With Data-Dependencies

```
// Potential side-effects, but we don't know.
def foo(): String = ???
def bar(a: String): String = ???
```

```
// Potential side-effects, but we don't know.
def foo(): String = ???
def bar(a: String): String = ???

// Data-dependency between functions calls.
val a = foo()
val b = bar(a)
```

# Functions: 3

Without Side-Effects
Without Data-Dependencies

```
// Effectful functions; encoded in signature.
def foo(): Task[String] = ???
def bar(): Task[String] = ???
```

```
// Effectful functions; encoded in signature.
def foo(): Task[String] = ???
def bar(): Task[String] = ???

// No data dependency between function calls.
val a = foo()
val b = bar()
```

```
// Effectful functions; encoded in signature.
def foo(): Task[String] = ???
def bar(): Task[String] = ???

// No data dependency between function calls.
val a = foo()
val b = bar()
val c = a.flatMap(_ => b) // sequential
```

```
// Effectful functions; encoded in signature.
def foo(): Task[String] = ???
def bar(): Task[String] = ???
```

```
// No data dependency between function calls.
val a = foo()
val b = bar()
val c = a.flatMap(_ => b) // sequential
val d = (a, b).mapN((a, b) => ...)

c.unsafeRunSync()
```

# Functions: 4

Without Side-Effects
With Data-Dependencies

```
// Effectful functions; encoded in signature.
def foo(): IO[String] = ???
def bar(a: String): IO[String] = ???
```

```
// Effectful functions; encoded in signature.
def foo(): IO[String] = ???
def bar(a: String): IO[String] = ???

// Data-dependency between functions calls.
val c = foo().flatMap(a => bar(a))
c.unsafeRunSync()
```

# Flashback

Applicative lets us embed function-like DSLs into our programs.

Applicative lets us perform global optimizations on the Abstract Syntax Tree of an EDSL.

# Roll-Ups are Functions

Idea: come up with an EDSL to model time series data fetching and constrain its usage to an Applicative interface.

Then, statically analyze the EDSL to batch and deduplicate issued queries before making the actual network calls.

```
// Independent computations

val resultAB = rollupAB(metricA(), metricB())
val resultAC = rollupAC(metricA(), metricC())
```

```
// Independent computations
val mA =
val resultAB = rollupAB(metricA(), metricB())
val resultAC = rollupAC(metricA(), metricC())
```

```
// Optimize: common subexpression elimination
val mA = metricA()
val resultAB = rollupAB(mA, metricB())
val resultAC = rollupAC(mA, metricC())
```

# Common subexpression elimination

In compiler theory, **common subexpression elimination** (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.

Contents

## Example

In the following code:

```
a = b * c + g;
d = b * c * e;
```

it may be worth transforming the code to:

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

if the cost of storing and retrieving `tmp` is less than the cost of calculating `b * c` an extra time.

# Let's Write an Optimizing EDSL Compiler

# Observations

Semantically-parallel Applicative instances don't bode well with Monad instances. If a type is a monadic, a lawful Applicative instance has to be sequential.

This is why Cats exposes a Parallel type-class, inspired by PureScript, which allows client code to choose between semantically-sequential and semantically-parallel Applicative instances.

# Future Work

# Future Work

## Static Analysis on Arrow Computations

# Call for Presentations

ionut.g.stan@gmail.com

# Questions!

# Thanks!