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

Consider the problem of building a network service that encapsulates business logic behind an API; a special case of this being a web-based application. Services of this kind often need to efficiently obtain and process data from a heterogeneous set of external sources. In the case of a web application, the service usually needs to access at least databases, and possibly other application-specific services that make up the distributed architecture of the system.

The *business logic* in this setting is the code that determines, for each request made using this service, what data to deliver as the result. In the case of a web application, the input is an HTTP request, and the output is a web page. Our goal is to have clear and 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). The cache memoises the results of previous data fetches, which provides not only performance benefits, but also consistency in the face of changes to the external data.
- We show that it isn't necessary for the programmer to use `Applicative` operators in order to benefit from concurrency in our framework, for two reasons: first, bulk monadic operations such as maps and filters use `Applicative` internally, which provides a lot of the benefit of `Applicative` concurrency for almost zero effort (Section 5.5), and secondly we can automatically translate code written using monadic style into `Applicative` in certain cases (Section 7).
- We have implemented this system at Facebook in a back-end service that contains over 200,000 lines of business logic. We present some preliminary results showing that our system running with production data efficiently optimises the data accesses. When running without our automatic concurrency, typical latencies were 51% longer (Section 9).

While our work is mostly focused on a setting in which all the operations of the DSL are data reads, we consider how to incorporate side-effecting operations in Section 9.3. Section 10 compares our design with other concurrent programming models.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9/14/09.
http://dx.doi.org/10.1145/2628136.2628144

2. Motivation

To motivate the design, we will present two use cases. The first is a typical web application, which needs to render a web page based on data fetched from one or more external sources. The second is a real-world use case from Facebook: a rule-engine for detecting certain types of content and taking actions based on it.

2.1 Example: rendering a blog

In this example we'll look at some code to render a blog, focusing on the part of the application that fetches and processes the data from the external data source (e.g. a database). The blog web page will consist of two panes:

- The main pane shows the most recent posts to the blog in date order.
- The side pane contains two sub-panes:
  - a list of the posts with the most page views ("popular posts"),
  - a list of topics and the number of posts in each topic.

Assuming a set of operations to fetch the necessary data, and a set of functions to actually render the HTML, the task is to write the code to collect the necessary data and call the rendering functions for each of the separate parts of the page. The goal is to write code that has two properties:

- It should be *modular*, so that new sections on the page can be added and removed without disturbing the rest of the code.
- It should execute *efficiently*, but without the programmer having to implement optimisations manually. In particular, we should be fetching as much data concurrently as possible.

Our framework allows both of these goals to be met; the code will be both maximally modular and maximally efficient (in terms of overlapping and batching external requests for data).

The example requires a bit of setup. First, some types:

```
data PostId       -- identifies a post
data Date         -- a calendar date
data PostContent -- the content of a post

data PostInfo = PostInfo
  { postId    :: PostId
  , postDate  :: Date
  , postTopic :: String
  }
```

A post on the blog is represented by two types: `PostInfo` and `PostContent`. `PostInfo` contains the metadata about the post: the date it was created, and its topic. The actual content of the post is represented by the abstract `PostContent` type.

Posts have an identifier that allows them to be fetched from the database, namely `PostId`. For the purposes of this example we will assume the simplest storage model possible: the storage performs no computation at all, so all sorting, joining, and so forth must be done by the client.

Our computation will be done in a monad called `Fetch`. The implementation of `Fetch` will be given later, but for this example all we need to know is that `Fetch` has instances of `Monad`, `Functor` and `Applicative`, and has the following operations for fetching data:

```
getPostIds     :: Fetch [PostId]
getPostInfo    :: PostId -> Fetch PostInfo
getPostContent :: PostId -> Fetch PostContent
getPostViews   :: PostId -> Fetch Int
```

`getPostIds` returns the identifiers of all the posts, `getPostInfo` retrieves the metadata about a particular post, `getPostContent` fetches the content of a post, and finally `getPostViews` returns a count of the number of page views for a post. Each of these operations needs to retrieve the data from some external source, perhaps one or more databases. Furthermore a database might be highly distributed, so there is no expectation that any two requests will be served by the same machine.

We assume a set of rendering functions, including.

```
renderPosts :: [(PostInfo,PostContent)] -> Html
renderPage  :: Html -> Html -> Html
```

`renderPosts` takes a set of posts and returns the corresponding HTML. Note that we need both the `PostInfo` and the `PostContent` to render a post. The `renderPage` function constructs the whole page given the HTML for the side pane and the main pane. We'll see various other functions beginning with `render`; the implementations of these functions aren't important for the example.

Now that the background is set, we can move on to the actual code of the example. We'll start at the top and work down; here is the top-level function, `blog`:

```
blog :: Fetch Html
blog = renderPage <$> leftPane <*> mainPane
```

`blog` generates a web page by calling `leftPane` and `mainPane` to generate the two panes, and then calling `renderPage` to put the results together. Note that we're using the `Applicative` combinators `<$>` and `<*>` to construct the expression: `leftPane` and `mainPane` are both `Fetch` operations because they will need to fetch data.

To make the main pane, we need to fetch all the information about the posts, sort them into date order, and then take the first few (say 5) to pass to `renderPosts`:

```
mainPane :: Fetch Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered =
        take 5 $
        sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)
```

Here `getAllPostsInfo` is an auxiliary function, defined as follows:

```
getAllPostsInfo :: Fetch [PostInfo]
getAllPostsInfo = mapM getPostInfo =<< getPostIds
```

As you might expect, to fetch all the `PostInfo`s we have to first fetch all the `PostId`s with `getPostIds`, and then fetch each `PostInfo` with `getPostInfo`.

The left pane consists of two sub-panes, so in order to construct the left pane we must render the sub-panes and put the result together by calling another rendering function, `renderSidePane`:

```
leftPane :: Fetch Html
leftPane = renderSidePane <$> popularPosts <*> topics
```

Next we'll look at the `popularPosts` sub-pane. In order to define this we'll need an auxiliary function, `getPostDetails`, which fetches both the `PostInfo` and the `PostContent` for a post:

```
getPostDetails :: PostId
               -> Fetch (PostInfo, PostContent)
getPostDetails pid =
 (,) <$> getPostInfo pid <*> getPostContent pid
```

Here is the code for `popularPosts`:

```
popularPosts :: Fetch Html
popularPosts = do
  pids <- getPostIds
  views <- mapM getPostViews pids
  let ordered =
        take 5 $ map fst $
        sortBy (flip (comparing snd))
               (zip pids views)
  content <- mapM getPostDetails ordered
  return $ renderPostList content
```

First we get the list of `PostId`s, and then the number of page views for each of these. The number of page views are used to sort the list; the value `ordered` is a list of the top five `PostId`s by page views. We can use this list to fetch the information about the posts that we need to render, by calling `getPostDetails` for each one, and finally the result is passed to `renderPostList` to render the list of popular posts.

Next the code for rendering the menu of topics:

```
topics :: Fetch Html
topics = do
  posts <- getAllPostsInfo
  let topiccounts =
        Map.fromListWith (+)
          [ (postTopic p, 1) | p <- posts ]
  return $ renderTopics topiccounts
```

Creating the list of topics is a matter of calculating a mapping from topic to the number of posts in that topic from the list of `PostInfo`s, and then passing that to `renderTopics` to render it.

This completes the code for the example. The code clearly expresses the functionality of the application, with no concession to performance. Yet we want it to execute efficiently too; there are two ways in which our framework will automatically improve the efficiency when this code is executed:

- **Concurrency**. A lot of the data fetching can be done concurrently. For example:
  - every time we use `mapM` with a data-fetching operation, there is an opportunity for concurrency.
  - we can compute `mainPane` and `leftPane` at the same time, and within `leftPane` we can compute `popularPosts` and `topics` at the same time.

  Our goal is to exploit all this inherent concurrency without the programmer having to lift a finger. The framework we will describe in this paper does exactly that: with the code as written, the data will be fetched in the pattern shown in Figure 1. The dotted lines indicate a *round* of data-fetching, where all the items in a round are fetched concurrently. There are three rounds:
  - `getPostIds` (needed by all three panes)
  - `getPostInfo` for all posts (needed by `mainPane` and `topics`), and `getPostViews` for all posts (needed by `popularPosts`).
  - `getPostContent` for each of the posts displayed in the main pane, and `getPostInfo` and `getPostContent` for each of the posts displayed in `popularPosts`.
- **Caching**. We made no explicit attempt to fetch each piece of data only once. For example, we are calling `getPostIds` three times. Remember the goal is to be *modular*: there is no global knowledge about what data is needed by each part of the page.

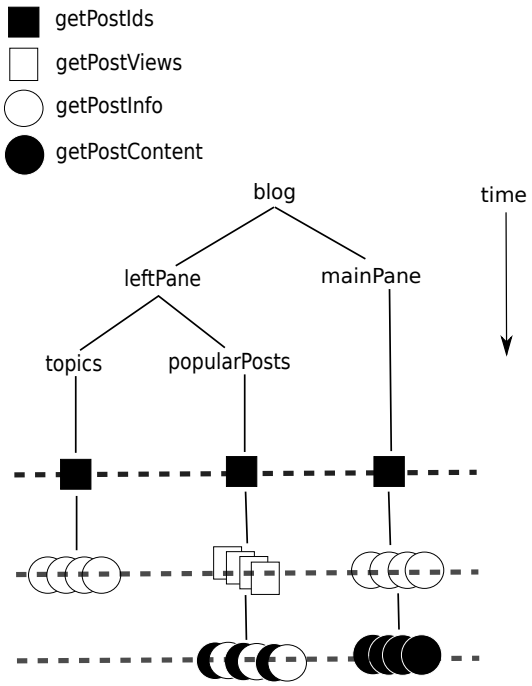

Figure 1. Data fetching in the blog example

Furthermore, even though we could reasonably predict that we need `getPostIds` in several places and so do it once up front, it is much harder to predict which `getPostContent` calls will be made: the main pane displays the five most recent posts, and the side pane displays the five most popular posts. There might well be overlap between these two sets, but to write the code to fetch the minimal set of `PostContent` would require destroying the modularity.

Our system uses caching to avoid fetching the same data multiple times, which lets the programmer keep the modularity in their code without worrying about duplicate data fetching. Furthermore, as we describe in Section 6, caching has important benefits beyond the obvious performance gains.

2.2 Example: a data-rich DSL

Our second use-case is a service inside the Facebook infrastructure that identifies spam, malware, and other types of undesirable content [11]. Every action that creates an item of content on the site results in a request to this service, and it is the job of the service to return a result indicating whether the content should be allowed or rejected[1]. The service runs on many machines, and each instance of the service runs the same set of business logic, which is typically modified many times per day.

As an example of the kind of calculations that our business logic needs to perform, consider this hypothetical expression fragment:

```
length (intersect (friendsOf x) (friendsOf y))
```

`length` is the usual list length operation, `intersect` takes the intersection of two lists, and `friendsOf` is a function that returns the list of friends of a user:

```
friendsOf :: UserId -> [UserId]
```

[1] This is a huge simplification, but will suffice for this paper.

The value of this expression is the number of friends that `x` and `y` have in common; this value tends to be a useful quantity in our business logic and is often computed.

This code fragment is an example of how we would like the business logic to look: clear, concise, and without any mention of implementation details.

Now, the `friendsOf` function needs to access a remote database in order to return its result. So if we were to implement this directly in Haskell, even if we hide the remote data access behind a pure API like `friendsOf`, when we run the program it will make two requests for data in series: first to fetch the friends of `x`, and then to fetch the friends of `y`. We ought to do far better than this: not only could we do these two requests concurrently, but in fact the database serving these requests (TAO, [14]) supports submitting several requests as a single batch, so we could submit both requests in a single unit.

The question is, how could we modify our language such that it supports an implementation that submits these two requests concurrently? The problem is not just one of exploring simple expressions like this; in general we might have to wait for the results of some data accesses before we can evaluate more of the expression. Consider this:

```
let
   numCommonFriends =
     length (intersect (friendsOf x) (friendsOf y))
in
if numCommonFriends < 2 && daysRegistered x < 30
   then ...
   else ...
```

Here `daysRegistered` returns the number of days that a user has been registered on the site.

So now, assuming that we want a lazy `&&` such that if the left side is `False` we don't evaluate the right side at all, then we cannot fetch the data for `daysRegistered` until we have the results of the two `friendsOf` calls.

Scaling this up, when we consider computing the result of a request that involves running a large amount of business logic, in general at any given time there might be many requests for data that could be submitted concurrently. Having fetched the results of those requests, the computation can proceed further, possibly along multiple paths, until it gets blocked again on a new set of data fetches.

Our solution is to build an abstraction using `Applicative` and `Monad` to support concurrent data access, which we describe in the next few sections. We will return in Section 5.3 to see how our DSL looks when built on top of the framework.

3. Concurrency monads

A concurrency monad embodies the fundamental notion of a computation that can pause and be resumed. The concurrency monad will be the foundation of the abstractions we develop in this paper. Here is its type:

```
data Fetch a = Done a | Blocked (Fetch a)
```

An operation of type `Fetch a` has either completed and delivered a value `a`, indicated by `Done`, or it is blocked (or paused), indicated by `Blocked`. The argument to `Blocked` is the computation to run to continue, of type `Fetch a`.

For reference, we give the definitions of the `Functor` and `Monad` type classes in Figure 2. The instances of `Functor` and `Monad` for `Fetch` are as follows:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Monad f where
  return :: a -> f a
  (>>=) :: f a -> (a -> f b) -> f b

ap :: (Monad m) => m (a -> b) -> m a -> m b
ap mf mx = do f <- mf; x <- mx; return (f x)
```

Figure 2. Definitions of `Functor`, `Applicative`, `Monad`, and `ap`

```
instance Functor Fetch where
  fmap f (Done x) = Done (f x)
  fmap f (Blocked c) = Blocked (fmap f c)

instance Monad Fetch where
  return = Done

  Done a    >>= k = k a
  Blocked c >>= k = Blocked (c >>= k)
```

In general, a computation in this monad will be a sequence of `Blocked` constructors ending in a `Done` with the return value. This is the essence of (cooperative) concurrency: for example, one could implement a simple round-robin scheduler to interleave multiple tasks by keeping track of a queue of blocked tasks, running the task at the front of the queue until it blocks again, and then returning it to the end of the queue.

Our monad isn't very useful yet. There are two key pieces missing: a way to introduce concurrency into a computation, and a way for a computation to say what data it is waiting for when it blocks. We will present these elaborations respectively in the next two sections. Following that, we will return to our motivating examples and show how the `Fetch` framework enables efficient and modular data-fetching.

4. Applicative concurrency

Concurrency monads have occurred in the literature several times. Scholz [10] originally introduced a concurrency monad based on a continuation monad, and then Claessen [2] used this as the basis for his Poor Man's Concurrency Monad. This idea was used by Li and Zdancewic [5] to implement scalable network services. A slightly different formulation but with similar functionality was dubbed the *resumption monad* by Harrison [4]. The resumption monad formulation was used in describing the semantics of concurrency by Swierstra and Altenkirch [12]. Our `Fetch` monad follows the resumption monad formulation. It is also worth noting that this idea is an instance of a *free monad* [1].

All these previous formulations of concurrency monads used some kind of *fork* operation to explicitly indicate when to create a new thread of control. In contrast, in this paper *there is no fork*. The concurrency will be implicit in the structure of the computations we write using this abstraction. To make it possible to build computations that contain implicit concurrency, we need to make `Fetch` an Applicative Functor [7]. For reference, the definition of the `Applicative` class is given in Figure 2 (omitting the `*>` and `<*` operators, which are not important for this paper).

Applicative Functors are a class of functors that may have effects that compose using the `<*>` operator. Morally, the class of Ap-

plicative Functors sits between Functors and Monads: every Monad is an Applicative Functor, but the reverse is not true. For historical reasons, `Applicative` is not currently a superclass of `Monad` in Haskell, although this is expected to change in the future.

An `Applicative` instance can be given for any `Monad`, simply by making `pure = return` and `<*> = ap` (Figure 2). However, for `Fetch` we want a custom `Applicative` instance that takes advantage of the fact that the arguments to `<*>` are independent, and uses this to introduce concurrency:

```
instance Applicative Fetch where
  pure = return

  Done g    <*> Done y    = Done (g y)
  Done g    <*> Blocked c = Blocked (g <$> c)
  Blocked c <*> Done y    = Blocked (c <*> Done y)
  Blocked c <*> Blocked d = Blocked (c <*> d)
```

This is the key piece of our design: when computations in `Fetch` are composed using the `<*>` operator, *both* arguments of `<*>` can be explored to search for `Blocked` computations, which creates the possibility that a computation may be blocked on multiple things simultaneously. This is in contrast to the monadic bind operator, `>>=`, which does not admit exploration of both arguments, because the right hand side cannot be evaluated without the result from the left.

For comparison, if we used `<*> = ap`, the standard definition for a `Monad`, we would get the following (refactored slightly):

```
instance Applicative Fetch where
  pure = return

  Done f    <*> x = f <$> x
  Blocked c <*> x = Blocked (c <*> x)
```

Note how only the first argument of `<*>` is inspected. The difference between these two will become clear if we consider an example: `Blocked (Done (+1)) <*> Blocked (Done 1)`. Under our `Applicative` instance this evaluates to:

```
 Blocked (Done (+1) <*> Done 1)
==>
 Blocked (Done (1 + 1))
```

whereas under the standard `Applicative` instance, the same example would evaluate to:

```
 Blocked (Done (+1) <*> Blocked (Done 1))
==>
 Blocked ((+1) <$> Blocked (Done 1))
==>
 Blocked (Blocked ((+1) <$> Done 1))
==>
 Blocked (Blocked (Done (1 + 1)))
```

If `Blocked` indicates a set of remote data fetches that must be performed (we'll see how this happens in the next section), then with our `Applicative` instance we only have to stop and fetch data once, whereas the standard instance has two layers of `Blocked`, so we would stop twice.

Now that we have established the basic idea, we need to elaborate it to do something useful; namely to perform multiple requests for data simultaneously.

5. Fetching data

In order to fetch some data, we need a primitive that takes a description of the data to fetch, and returns the data itself. We will call this operation `dataFetch`:

```
dataFetch :: Request a -> Fetch a
```

where `Request` is an application-specific type that specifies requests; a value of type `Request a` is an instruction that the system can use to fetch a value of type `a`. For now the `Request` type is a concrete but unspecified type; we will show how to instantiate this for our blog example in Section 5.2, and we outline how to abstract the framework over the request type in Section 9.

How can we implement `dataFetch`? One idea is to elaborate the `Blocked` constructor to include a request:

```
data Fetch a
  = Done a
  | forall r . Blocked (Request r) (r -> Fetch a)
```

This works for a single request, but quickly runs into trouble when we want to block on *multiple* requests because it becomes hard to maintain the connections between multiple result types `r` and their continuations.

We solve this problem by storing results in mutable references. This requires two changes. First we encapsulate the request and the place to store the result in an existentially quantified `BlockedRequest` type:

```
data BlockedRequest =
  forall a . BlockedRequest (Request a)
                            (IORef (FetchStatus a))
```

(a `forall` outside the constructor definition is Haskell's syntax for an existentially-quantified type variable). `IORef` is Haskell's mutable reference type, which supports the following operations for creation, reading and writing respectively:

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

The `FetchStatus` type is defined as follows:

```
data FetchStatus a
  = NotFetched
  | FetchSuccess a
```

Before the result is available, the `IORef` contains `NotFetched`. When the result is available, it contains `FetchSuccess`. As we will see later, using an `IORef` here also makes it easier to add caching to the framework.

The use of `IORef` requires that we layer our monad on top of the `IO` monad. In practice this isn't a drawback, because the `IO` monad is necessary in order to perform the actual data fetching, so it will be available when executing a computation in the `Fetch` monad anyway. The `IO` monad will not be exposed to user code.

Considering that we will need computations that can block on multiple requests, our monad now also needs to collect the set of `BlockedRequest` associated with a blocked computation. A list would work for this purpose, but it suffers from performance problems due to nested appends, so instead we will use Haskell's `Seq` type, which supports logarithmic-time append.

With these two modifications (adding `IO` and attaching `Seq BlockedRequest` to the `Blocked` constructor), the monad now looks like this:

```
data Result a
  = Done a
  | Blocked (Seq BlockedRequest) (Fetch a)

newtype Fetch a = Fetch { unFetch :: IO (Result a) }
```

```
instance Applicative Fetch where
  pure = return

  Fetch f <*> Fetch x = Fetch $ do
    f' <- f
    x' <- x
    case (f',x') of
      (Done g,         Done y       ) -> return (Done (g y))
      (Done g,         Blocked br c ) -> return (Blocked br (g <$> c))
      (Blocked br c,   Done y       ) -> return (Blocked br (c <*> return y))
      (Blocked br1 c, Blocked br2 d) -> return (Blocked (br1 <> br2) (c <*> d))
```

Figure 3. Applicative instance for `Fetch`

```
instance Monad Fetch where
  return a = Fetch $ return (Done a)

  Fetch m >>= k = Fetch $ do
    r <- m
    case r of
      Done a       -> unFetch (k a)
      Blocked br c -> return (Blocked br (c >>= k))
```

and the `Applicative` instance is given in Figure 3. Note that in the case where both arguments to `<*>` are `Blocked`, we must combine the sets of blocked requests from each side.

Finally we are in a position to implement `dataFetch`:

```
dataFetch :: Request a -> Fetch a
dataFetch request = Fetch $ do
  box <- newIORef NotFetched            -- (1)
  let br = BlockedRequest request box   -- (2)
  let cont = Fetch $ do                 -- (3)
        FetchSuccess a <- readIORef box -- (4)
        return (Done a)                 -- (5)
  return (Blocked (singleton br) cont)  -- (6)
```

Where:

- Line 1 creates a new `IORef` to store the result, initially containing `NotFetched`.
- Line 2 creates a `BlockedRequest` for this request.
- Lines 3–5 define the continuation, which reads the result from the `IORef` and returns it in the monad. Note that the contents of the `IORef` is assumed to be `FetchSuccess a` when the continuation is executed. It is an internal error of the framework if this is not true, so we don't attempt to handle the error condition here.
- Line 6: `dataFetch` returns `Blocked` in the monad, including the `BlockedRequest`.

5.1 Running a computation

We've defined the `Fetch` type and its `Monad` and `Applicative` instances, but we also need a way to *run* a `Fetch` computation. Clearly the details of how we actually fetch data are application-specific, but there's a standard pattern for running a computation that works in all settings.

The application-specific data-fetching can be abstracted as a function `fetch`:

```
fetch :: [BlockedRequest] -> IO ()
```

The job of `fetch` is to fill in the `IORef` in each `BlockedRequest` with the data fetched. Ideally, `fetch` will take full advantage of concurrency where possible, and will batch together requests for data from the same source. For example, multiple HTTP requests could be handled by a pool of connections where each connection processes a pipelined batch of requests. Our actual implementation at Facebook has several data sources, corresponding to various internal services in the Facebook infrastructure. Most have asynchronous APIs but some are synchronous, and several of them support batched requests. We can fetch data from all of them concurrently.

Given `fetch`, the basic scheme for running a `Fetch` computation is as follows:

```
runFetch :: Fetch a -> IO a
runFetch (Fetch h) = do
  r <- h
  case r of
    Done a -> return a
    Blocked br cont -> do
      fetch (toList br)
      runFetch cont
```

This works as follows. First, we run the `Fetch` computation. If the result was `Done`, then we are finished; return the result. If the result was `Blocked`, then fetch the data by calling `fetch`, and then run the continuation from the `Blocked` constructor by recursively invoking `runFetch`.

The overall effect is to run the computation in stages that we call *rounds*. In each round `runFetch` performs as much computation as possible and then performs all the data fetching concurrently. This process is repeated until the computation returns `Done`.

By performing as much computation as possible we maximise the amount of data fetching we can perform concurrently. This makes good use of our network resources, by providing the maximum chance that we can batch multiple requests to the same data source, but it might not be the optimal scheme from a latency perspective; we consider alternatives in Section 11.

Our design does not impose a particular concurrency strategy on the data sources. The implementation of `fetch` has complete freedom to use the most appropriate strategy for executing the requests it is given. Typically that will involve a combination of batching requests to individual data sources, and performing requests to multiple data sources concurrently with each other using Haskell's existing concurrency mechanisms.

5.2 Example: blog

In this section we will instantiate our framework for the blog example described in Section 2.1, and show how it delivers automatic concurrency.

First, we need to define the `Request` type. Requests are parameterised by their result type, and since there will be multiple requests with different result types, a `Request` must be a GADT [9]. Here is the `Request` type for our blog example:

```
data Request a where
  FetchPosts       :: Request [PostId]
  FetchPostInfo    :: PostId -> Request PostInfo
  FetchPostContent :: PostId -> Request PostContent
  FetchPostViews   :: PostId -> Request Int
```

Next we need to provide implementations for the data-fetching operations (`getPostIds` etc.), which are simply calls to `dataFetch` passing the appropriate `Request`:

```
getPostIds     = dataFetch FetchPosts
getPostInfo    = dataFetch . FetchPostInfo
getPostContent = dataFetch . FetchPostContent
getPostViews   = dataFetch . FetchPostViews
```

Now, if we provide a dummy implementation of `fetch` that simulates a remote data source and prints out requests as they are made[2], we do indeed find that the requests are made in three rounds as described in Section 2.1. A real implementation of `fetch` would perform the requests in each round concurrently.

5.3 Example: Haxl

In Section 2.2 we introduced our motivation for designing the applicative concurrency abstraction. Our implementation is called `Haxl`, and we will describe it in more detail in Section 9.1. Here, we briefly return to the original example to show how to implement it using `Fetch`.

The example we used was this expression:

```
length (intersect (friendsOf x) (friendsOf y))
```

How does this look when used with our `Fetch` monad? Any operation that may fetch data must be a `Fetch` operation, hence

```
friendsOf :: UserId -> Fetch [UserId]
```

while `length` and `intersect` are the usual pure functions. So to write the expression as a whole we need to lift the pure operations into the `Applicative` world, like so:

```
length <$> intersect' (friendsOf x) (friendsOf y)
  where intersect' = liftA2 intersect
```

This is just one way we could write it, there are many other equivalent alternatives. As we shall see in Section 7, it is also acceptable to use the plain do-notation, together with a source-to-source transformation that turns do-notation into `Applicative` operations:

```
do a <- friendsOf x
   b <- friendsOf y
   return (length (intersect a b))
```

In fact, this is the style we advocate for users of our DSL.

5.4 Semantics of `Fetch`

It's worth pondering on the implications of what we have done here. Arguably we broke the rules: while the `Applicative` laws do hold for `Fetch`, the documentation for `Applicative` also states that if a type is also a `Monad`, then its `Applicative` instance should satisfy `pure = return` and `<*> = ap`. This is clearly not the case for our `Applicative` instance. But in some sense, our intentions are pure: the goal is for code written using `Applicative` to execute more efficiently, not for it to give a different answer than when written using `Monad`.

Our justification for this `Applicative` instance is based on more than its literal definition. We intend `dataFetch` to have certain properties: it should not be observable to the programmer writing code using `Fetch` whether their `dataFetch` calls were performed concurrently or sequentially, or indeed in which order they were performed, the results should be the same. Therefore, `dataFetch` should not have any observable side-effects—all our requests must be read-only. To the user of `Fetch` it is *as if* the `Applicative` instance is the default `<*> = ap`, except that the code runs more efficiently, and for this to be the case we must restrict ourselves to read-only requests (although we return to this question and consider side-effects again in Section 9.3).

Life is not quite that simple, however, since we are reading data from the outside world, and the data may change between calls to `dataFetch`. The programmer might be able to observe a change in the data and hence observe an ordering of `dataFetch` operations. Our approach is to close this loophole as far as we can: in Section 6 we add a cache to the system, which will ensure that identical requests always return the same result within a single run of `Fetch`. Technically we can argue that `runFetch` is in the `IO` monad and therefore we are justified in making a non-deterministic choice for the ordering of `dataFetch` operations, but in practice we find that for the majority of applications this technicality is not important: we just write code as if we are working against a snapshot of the external data.

If we actually did have access to an unchanging snapshot of the remote data, then we could make a strong claim of determinism for the programming model. Of course that's not generally possible when there are multiple data sources in use, although certain individual data sources do support access to a fixed snapshot of their data; one example is Datomic[3].

5.5 Bulk operations: `mapM` and `sequence`

In our example blog code we used the combinators `mapM` and `sequence` to perform bulk operations. As things stand in Haskell today, these functions are defined using monadic bind, for example `sequence` is defined in the Haskell 2010 Report as

```
sequence :: Monad m => [m a] -> m [a]
sequence =  foldr mcons (return [])
 where mcons p q = do x <- p; y <- q; return (x:y)
```

Unfortunately, because this uses monadic bind rather than Applicative `<*>`, in our framework it will serialise the operations rather than perform them concurrently. Fortunately `sequence` doesn't require monadic bind; `Applicative` is sufficient [7], and indeed the the `Data.Traversable` module provides an equivalent that uses `Applicative`: `sequenceA`. Similarly, `traverse` is the `Applicative` equivalent of `mapM`. Nevertheless, Haskell programmers tend to be less familiar with the `Applicative` equivalents, so in our EDSL library we map `sequence` to `sequenceA` and `mapM` to `traverse`, so that client code can use these well-known operations and obtain automatic concurrency.

In due course when `Applicative` is made a superclass of `Monad`, the `Applicative` versions of these functions will become the defaults, and our workaround can be removed without changing the client code or its performance.

6. Adding a cache

In Section 2.1 we identified two ways that the framework can provide automatic performance benefits for the application. So far we have demonstrated the first, namely exploiting implicit concurrency. In this section we turn our attention to the second: avoiding duplicate requests for data.

[2] Sample code is available at `https://github.com/simonmar/haxl-icfp14-sample-code`

[3] `http://www.datomic.com/`

The solution is not surprising, namely to add caching. However, as we shall see, the presence of a cache provides some rather nice benefits in addition to the obvious performance improvements.

Recall that data is fetched using `dataFetch`:

```
dataFetch :: Request a -> Fetch a
```

Caching amounts to memoising this operation, such that the second time it is called with a request that has been previously issued, it returns the result from the original request. Not only do we gain performance by not repeating identical data-fetches, as mentioned in Section 5.4 the programmer can rely on identical requests returning the same results, which provides consistency within a single `Fetch` computation in the face of data that might be changing.

We also gain the ability to do some source-to-source transformations. For example, common subexpression elimination:

```
  do x <- N; M
 ==>
  do x <- N; M[return x/N]
```

Where `M` and `N` stand for arbitrary `Fetch` expressions, This holds provided `dataFetch` is the only way to do I/O in our framework, and all `dataFetch` requests are cached.

6.1 Implementing the cache

Let's consider how to add a cache to the system. In order to store a mapping from requests to results, we need the following API:

```
data DataCache

lookup :: Request a -> DataCache -> Maybe a
insert :: Request a -> a -> DataCache -> DataCache
```

If we want to use an existing efficient map implementation, we cannot implement this API directly because its type-correctness relies on the correctness of the map implementation, and the `Eq` and `Ord` instances for `Request`. But if we trust these, Haskell provides an unsafe back-door, `unsafeCoerce`, that lets us convey this promise to the type system. The use of unsafe features to implement a purely functional API is common practice in Haskell; often the motivation is performance, but here it is the need to maintain a link between two types in the type system.

A possible implementation is as follows:

```
newtype DataCache =
  DataCache (forall a . HashMap (Request a) a)
```

The contents of a `DataCache` is a mapping that, for all types `a`, maps things of type `Request a` to things of type `a`. The invariant we require is that a key of type `Request a` is either not present in the mapping, or maps to a value of type `a`. We will enforce the invariant when an element is inserted into the `Map`, and assume it when an element is extracted. If the `Map` is correctly implemented, then our assumption is valid.

Note that we use a `HashMap` rather than a plain `Map`. This is because `Map` requires the key type to be an instance of the `Ord` class, but `Ord` cannot be defined for all `Request a` because it would entail comparing keys of different types. On the other hand, `HashMap` requires `Eq` and `Hashable`, both of which can be straightforwardly defined for `Request a`, the former using a standalone deriving declaration:

```
deriving instance Eq (Request a)
```

and the latter with a hand-written `Hashable` instance (see the sample code[4]).

[4] `https://github.com/simonmar/haxl-icfp14-sample-code`

Looking up in the cache is simply a `lookup` in the `Map`:

```
lookup :: Request a -> DataCache -> Maybe a
lookup key (DataCache m) = Map.lookup key m
```

This works because we have already declared that the `Map` in a `DataCache` works for all types `a`. The `insert` operation is where we have to make a promise to the type system:

```
insert :: Request a -> a -> DataCache -> DataCache
insert key val (DataCache m) =
  DataCache $ unsafeCoerce (Map.insert key val m)
```

We can insert a key/value pair into the `Map` without any difficulty. However, that results in a `Map` instantiated at a *particular* type `a` (the type of `val` passed to `insert`), so in order to get back a `Map` that works for any `a` we need to apply `unsafeCoerce`. The `unsafeCoerce` function has this type:

```
unsafeCoerce :: forall a b . a -> b
```

Therefore, applying `unsafeCoerce` to the `Map` allows it to be generalised to the type required by `DataCache`.

Now we have a cache that can store a type-safe mapping from requests to results. We will need to plumb this around the `Monad` to pass it to each call to `dataFetch` so that we can check the cache for a previous result. However, this won't be enough: consider what happens when we make two identical requests in the same *round*: there won't be a cached result, but nevertheless we want to ensure that we only make a single request and use the same result for both `dataFetch` calls. Indeed, this happens several times in our blog example: the first round issues three calls to `getPostIds`, for example.

In `dataFetch` we need to distinguish three different cases:

1. The request has not been encountered before: we need to create a `BlockedRequest`, and block.
2. The request has already been fetched: we can return the cached result and continue.
3. The request has been encountered in the current round but not yet fetched: we need to block, but not create a new `BlockedRequest` since it will already have been added to the set of requests to fetch elsewhere.

The key idea is that in the third case we can share the `IORef (FetchStatus a)` from the `BlockedRequest` that was created the first time the request was encountered. Hence, all calls to `dataFetch` for a given request will automatically share the same result. How can we find the `IORef` for a request? *We store it in the cache*.

So instead of storing only results in our `DataCache`, we need to store `IORef (FetchStatus a)`. This lets us distinguish the three cases above:

1. The request is not in the `DataCache`.
2. The request is in the `DataCache`, and the `IORef` contains `FetchSuccess a`.
3. The request is in the `DataCache`, and the `IORef` contains `NotFetched`.

This implies that we must add an item to the cache as soon as the request is issued; we don't wait until the result is available. Filling in the details, our `DataCache` now has the following API:

```
data DataCache

lookup :: Request a -> DataCache
       -> Maybe (IORef (FetchStatus a))
```

```
dataFetch :: Request a -> Fetch a
dataFetch req = Fetch $ \ref -> do
  cache <- readIORef ref
  case lookup req cache of
    Nothing -> do
      box <- newIORef NotFetched
      writeIORef ref (insert req box cache)
      let br = BlockedRequest req box
      return (Blocked (singleton br) (cont box))
    Just box -> do
      r <- readIORef box
      case r of
        FetchSuccess result ->
          return (Done result)
        NotFetched ->
          return (Blocked Seq.empty (cont box))
 where
  cont box = Fetch $ \ref -> do
    FetchSuccess a <- readIORef box
    return (Done a)
```

Figure 4. `dataFetch` implementation with caching

```
insert :: Request a -> IORef (FetchStatus a)
       -> DataCache -> DataCache
```

(the implementation is the same). The cache itself needs to be stored in an `IORef` and passed around in the monad; `Fetch` now has this definition:

```
newtype Fetch a = Fetch {
  unFetch :: IORef DataCache -> IO (Result a) }
```

The alterations to the `Monad` and `Applicative` instances are straightforward, so we omit them here.

The definition of `dataFetch` is given in Figure 4, The three cases identified earlier are dealt with in that order:

1. If the request is not in the cache, then we create a new `IORef` for the result (initially containing `NotFetched`) and add that to the cache. Then we create a `BlockedRequest`, and return `Blocked` in the monad, with a continuation that will read the result from the `IORef` we created.
2. If the request is in the cache, then we check the contents of the `IORef`. If it contains `FetchSuccess result`, then we have a cached result, and `dataFetch` returns `Done` immediately (it doesn't block).
3. If the contents of the `IORef` is `NotFetched`, then we return `Blocked`, but with an empty set of `BlockedRequests`, and a continuation that will read the result from the `IORef`.

6.2 Cache Persistence and Replaying

Within a single `runFetch`, the cache only accumulates information, and never discards it. In the use-cases we have described, this is not a problem: requests to a network-based service typically take a short period of time to deliver the result, after which we can discard the cache. During a computation we don't want to discard any cached data, because the programmer might rely on the cache for consistency.

We have found that the cache provides other benefits in addition to the ones already described:

- at the end of a `Fetch` computation, the cache is a complete record of all the requests that were made, and the data that was fetched. Re-running the computation with the fully populated cache is guaranteed to give the same result, and will not fetch any data. So by persisting the cache, we can replay computations for the purposes of fault diagnosis or profiling. When the external data is changing rapidly, being able to reliably reproduce past executions is extremely valuable.
- We can store things in the cache that are not technically remote data fetches, but nevertheless we want to have a single deterministic value for. For example, in our implementation we cache the current time: within a `Fetch` computation the current time is a constant. We can also memoise whole `Fetch` computations by storing their results in the cache.

7. Automatic Applicative

Our `Fetch` abstraction requires the programmer to use the operations of `Applicative` in order to benefit from concurrency. While these operations are concise and expressive, many programmers are more comfortable with monadic notation and prefer to use it even when `Applicative` is available. Furthermore, we don't want to penalise code that uses monadic style: it should be automatically concurrent too. Our monad is commutative, so we are free to re-order operations at will, including replacing serial `>>=` with concurrent `<*>`.

In general, the transformation we want to apply is this:

```
  do p <- A; q <- B; ...
==>  {- if no variable of p is a free variable of B -}
  do (p,q) <- (,) <$> A <*> B
```

for patterns `p` and `q` and expressions `A` and `B`. The transformation can be applied recursively, so that long sequences of independent statements in do-notation can be automatically replaced by `Applicative` notation.

At the time of writing, the transformation is proposed but not implemented in GHC; it is our intention to implement it as an optional extension (because it is not necessarily valid for every `Applicative` instance). In our `Haxl` implementation we currently apply this transformation as part of the automatic translation of our existing DSL into Haskell.

8. Exceptions

Handling failure is an important part of a framework that is designed to retrieve data from external sources. We have found that it is important for the application programmer to be able to handle failure, particularly transient failures that occur due to network problems or outages in external services. In these cases the programmer typically wants to choose between having the whole computation fail, or substituting a conservative default value in place of the data requested.

We need to consider failure in two ways: first, the way in which exceptions propagate in the monad, and second, how failure is handled at the data-fetching layer. We'll deal with these in order.

8.1 Exceptions in `Fetch`

First, we add explicit exception support to our monad. We need to add one constructor to the `Result` type, `Throw`, which represents a thrown exception:

```
data Result a
  = Done a
  | Blocked (Seq BlockedRequest) (Fetch a)
  | Throw SomeException
```

The `SomeException` type is from Haskell's `Control.Exception` library and represents an arbitrary exception [6]. To throw an ex-

ception we need to convert it to a `SomeException` and return it with `Throw`:

```
throw :: Exception e => e -> Fetch a
throw e = Fetch $ \_ ->
  return (Throw (toException e))
```

The `Monad` instance for `Fetch` with the `Throw` constructor is as follows:

```
instance Monad Fetch where
  return a = Fetch $ \ref -> return (Done a)

  Fetch m >>= k = Fetch $ \ref -> do
    r <- m ref
    case r of
      Done a       -> unFetch (k a) ref
      Blocked br c -> return (Blocked br (c >>= k))
      Throw e      -> return (Throw e)
```

and Figure 5 gives the `Applicative` instance. It is straightforward except for one case: in `<*>`, where the left side returns `Blocked` and the right side returns `Throw`, we *must not propagate the exception yet*, and instead we must return a `Blocked` computation. The reason is that we don't yet know whether the left side will throw an exception when it becomes unblocked; if it does throw an exception, then that is the exception that the computation as a whole should throw, and not the exception from the right argument of `<*>`. If we were to throw the exception from the right argument of `<*>` immediately, the result would be non-determinism: the exception that gets thrown depends on whether the left argument blocks.

We also need a `catch` function:

```
catch :: Exception e
      => Fetch a -> (e -> Fetch a) -> Fetch a

catch (Fetch h) handler = Fetch $ \ref -> do
  r <- h ref
  case r of
    Done a -> return (Done a)
    Blocked br c ->
      return (Blocked br (catch c handler))
    Throw e -> case fromException e of
      Just e' -> unFetch (handler e') ref
      Nothing -> return (Throw e)
```

As with `catch` in the `IO` monad, our `catch` catches only exceptions of the type expected by the handler (the second argument to `catch`). The function `fromException` returns `Just e'` if the exception can be coerced to the appropriate type, or `Nothing` otherwise. The interesting case from our perspective is the `Blocked` case, where we construct the continuation by wrapping a call to `catch` around the inner continuation.

8.2 Exceptions in `dataFetch`

When a failure occurs in a data fetching operation, it must be thrown as an exception to the caller of `dataFetch`. We need to program this propagation explicitly, because the data is being fetched in the top-level `runFetch` loop, outside the context of the `Fetch` computation that called `dataFetch`.

We propagate an exception in the same way that we communicate the result of the data fetch: via the `IORef` that stores the result. So we modify the `FetchStatus` type to include the possibility that the fetch failed with an exception:

```
data FetchStatus a
  = NotFetched
  | FetchSuccess a
  | FetchFailure SomeException
```

and we also modify `dataFetch` to turn a `FetchFailure` into a `Throw` after the fetch has executed (these modifications are straightforward, so we omit the code here).

This is all the support we need for exceptions. There is one pitfall: we found in our real implementation that some care is needed in the implementation of a data source to ensure that an exception is properly reported as a `FetchFailure` and not just thrown by the data source; the latter causes the whole `Fetch` computation to be aborted, since the exception is thrown during the call to `fetch` in `runFetch`.

9. Implementation and evaluation

The basics of our use-case at Facebook were introduced in Section 2.2. Essentially it is a network-based service that is used to detect and eliminate spam, malware, and other undesirable content on Facebook. There are about 600 different kinds of request, all implemented by a body of Haskell code of approximately 200,000 lines; this was automatically translated into Haskell from our previous in-house DSL, FXL.

The system can be viewed as a rule-engine, where rules are `Fetch` computations. Each request runs a large set of rules and aggregates the results from all the rules. Rules are often (but not always) short, and most of them fetch some external data. In our system we run all the rules for a request using `sequence`; this has the effect of executing all the rules concurrently.

We will give an outline of our implementation in the next section, and then present some preliminary results.

9.1 Implementation

In the earlier description, the implementation of the `Fetch` monad depended on the `Request` type, because the monad carries around a `DataCache` that stores `Requests`, and the `dataFetch` operation takes a `Request` as an argument. This is straightforward but somewhat inconvenient, because we want to have the flexibility to add new data sources in a modular way, without modifying a single shared `Request` type. Furthermore, we want to be able to build and test data sources independently of each other, and to test the framework against "mock" versions of the data sources that don't fetch data over the wire.

To gain this flexibility, in our implementation we abstracted the core framework over the data sources and request types. Space limitations preclude a full description of this, but the basic idea is to use Haskell's `Typeable` class so that we can store requests of arbitrary type in the cache. The `dataFetch` operation has this type:

```
dataFetch :: (DataSource req, Request req a)
          => req a -> Fetch a
```

where `Request` is a package of constraints including `Typeable`, and `DataSource` is defined like this:

```
class DataSource req where
  fetch :: [BlockedFetch req] -> PerformFetch

data PerformFetch
  = SyncFetch  (IO ())
  | AsyncFetch (IO () -> IO ())
```

A data source is coupled to the type of requests that it serves, so for each request type there must be an instance of `DataSource` that defines how those requests are fetched. The `fetch` method takes

```
instance Applicative Fetch where
  pure = return

  Fetch f <*> Fetch x = Fetch $ \ref -> do
    f' <- f ref
    x' <- x ref
    case (f',x') of
      (Done g,         Done y       ) -> return (Done (g y))
      (Done g,         Blocked br c ) -> return (Blocked br (g <$> c))
      (Done g,         Throw e      ) -> return (Throw e)
      (Blocked br c,   Done y       ) -> return (Blocked br (c <*> return y))
      (Blocked br1 c, Blocked br2 d) -> return (Blocked (br1 <> br2) (c <*> d))
      (Blocked br c,   Throw e      ) -> return (Blocked br (c <*> throw e))
      (Throw e,        _            ) -> return (Throw e)
```

Figure 5. Applicative instance for `Fetch` with exceptions

a list of `BlockedRequests` containing requests that belong to this data source (the `BlockedRequest` type is now parameterised by the type of the request that it contains). The job of `fetch` is to fetch the data for those requests; it can do that synchronously or asynchronously, indicated by the `PerformFetch` type. An `AsyncFetch` is a function that takes as an argument the `IO` operation to perform *while the data is being fetched*. The idea is that when fetching data from multiple sources we wrap all the asynchronous fetches around a sequence of the synchronous fetches:

```
scheduleFetches :: [PerformFetch] -> IO ()
scheduleFetches fetches = asyncs syncs
 where
  asyncs = foldr (.) id [f | AsyncFetch f <- fetches]
  syncs  = sequence_ [io | SyncFetch io <- fetches]
```

In our implementation, most data sources are asynchronous. Maximal concurrency is achieved when at most one data source in a given round is synchronous, which is the case for the vast majority of our fetching rounds. When there are multiple synchronous data sources we could achieve more concurrency by using Haskell's own concurrency mechanisms; this is something we intend to explore in the future.

9.2 Results

To evaluate how well our system exploits concurrency, we ran a random sample of 10,000 actual requests for a single common request type. We measured the number of data fetches performed by each request (not including those that were served from the cache), the number of *rounds* (batches of fetches performed concurrently), and the total end-to-end processing time of each request. Figure 6 gives the results, in the form of histograms of the number of requests against fetches, rounds, and total time (latency). Note that the number of requests on the Y-axis is a log scale. In the histogram of fetches, the buckets are 5 wide, so for example the first bar represents the number of requests with 10–15 data fetches (there were no requests that performed fewer than 10 fetches). The histogram of rounds has integral buckets, and the time histogram has buckets of 20ms.

Figure 7 gives the 50^{th} (median), 95^{th}, and 99^{th} percentiles, and the maximum value, for each of fetches, rounds, and time. Note that the figures for each column were calculated by sorting the requests by fetches, rounds, and time respectively. It is not necessarily the case that the request that performed the maximum number of fetches is the same request that took the maximum number of rounds or the longest time.

We can see that 95% of our 10,000 requests require at most 4 rounds of fetching (median 3), 95% perform at most 27 data fetches (median 18), and 95% run in at most 26.3ms (median 9.5ms). There is a long tail, however, with some requests requiring more than 2000 data fetches. A few requests took an inordinately long time to run (the longest was 2.2s), and this turned out to be because one particular data fetch to another service took a long time.

The second table in Figure 7 shows for comparison what happens when we disable concurrency—this was achieved by making `(<*>) = ap`, so that `<*>` no longer batches together the fetches from both of its arguments (caching was still enabled, however). We can see that the number of rounds is equal to the number of fetches, as expected. The experiments were run against production data, so there are minor differences in the number of fetches between the two runs in Figure 7, but we can see that the effect on total runtime is significant, increasing the median time for a request by 51%. One extreme example is the request that required 2793 fetches, which increased from 220ms to 1.3s with concurrency disabled. Concurrency had no effect on the pathological data fetches, so the maximum time was unchanged at 2.2s.

9.2.1 Discussion

We have shown that the automatic concurrency provided by our framework has a sizeable impact on latency for requests in our system, but is it enough? Our existing FXL-based system performs similar data-fetching optimisations, but it does so using a special-purpose interpreter, whereas our Haskell version is implemented in libraries without modifying the language implementation.

Our workload is primarily I/O bound, so although Haskell is far faster than FXL at raw compute workloads, this has little effect on comparisons between our two systems. Thus we believe that executing data fetches concurrently is the most important factor affecting performance, and if the Haskell system were less able to exploit concurrency that would hinder its performance in these benchmarks. At the time of writing we have only preliminary measurements, but performance of the two systems does appears to be broadly similar, and we have spent very little time optimising the Haskell system so far.

It is also worth noting that the current workload is I/O bound partly because compute-heavy tasks have historically been offloaded to C++ code rather than written in FXL, because using FXL would have been too slow. In the Haskell version of our system we have reimplemented some of this functionality natively in Haskell, because its performance is more than adequate for compute tasks, and the Haskell code is significantly cleaner and safer. We believe that being able to implement compute tasks directly in Haskell will empower the users of our DSL to solve problems that they couldn't previously solve without adding C++ primitives to the language implementation.

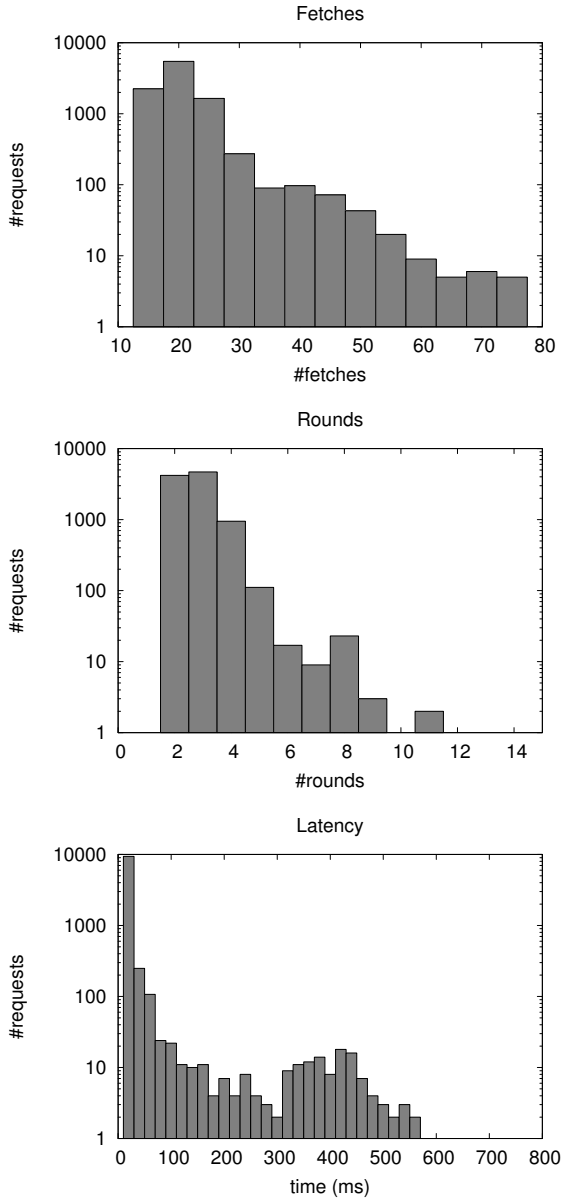

Figure 6. Results

Concurrency	**Fetches**	**Rounds**	**Time (ms)**
50%	18	3	9.5
95%	27	4	26.3
99%	47	5	356.5
Max	2793	11	2200.0

No concurrency	**Fetches**	**Rounds**	**Time (ms)**
50%	17	17	14.4
95%	27	27	36.1
99%	46	46	381.4
Max	2792	2792	2200.0

Figure 7. Summary results, with and without concurrency

9.3 Using Applicative Concurrency with Side-effects

As described, our framework has no side-effects except for reading, for good reason: operations in `Fetch` may take place in any order (Section 5.4). However, side-effects are important. For example, a web application needs to take actions based on user input, and it might need to generate some statistics that get stored. Our implementation at Facebook has various side effects, including storing values in a separate `memcache` service, and incrementing shared counters.

One safe way to perform side effects is to return them from `runFetch`, and perform them afterwards. Indeed, this is exactly the way that side effects are typically performed when using Software Transactional Memory (`STM`) in Haskell.

Sometimes it is convenient to allow side-effects as part of the `Fetch` computation itself. This is fine as long as it is *not possible to observe the side-effect with a* `Fetch` *operation*, which would expose the ordering of operations to the user. But this is quite flexible: we can, for example, have a write-only instance of `Fetch` that allows write operations to benefit from concurrency (obviously, the cache is not necessary for this), or we can have side-effects that cannot be observed, such as accumulating statistics.

10. Comparison and related work

Probably the closest relatives to the `Fetch` framework are the family of *async* programming models that have been enjoying popularity recently in several languages: F# [13], C#, OCaml [15], Scala [3], and Clojure[5].

A common trait of these programming models is that they are based on a concurrency-monad-like substrate; they behave like lightweight threads with cooperative scheduling. When a computation is suspended, its continuation is saved and re-executed later. These frameworks are typically good for scheduling large numbers of concurrent I/O tasks, because they have lower overhead than the heavyweight threads of their parent languages.

In contrast with the `Fetch` framework, the async style has an explicit fork operation, in the form of an asynchronous method call that returns an object that can later be queried for the result. For example, in C# a typical sequence looks like this:

```
Task<int> a = getData();
int y = doSomethingElse();
int x = await a;
```

The goal in this pattern is to perform `getData()` concurrently with `doSomethingElse()`. The effects of `getData()` will be interleaved with those of `doSomethingElse()`, although the degree of non-determinism is tempered somewhat by the use of cooperative scheduling.

Ignoring non-determinism, in our system this could be written

```
do [x,y] <- sequence [getData,doSomethingElse]
   ...
```

making it clear that `getData` and `doSomethingElse` are executed together.

A similar style is available in F# and C# using `Async.Parallel` and `Task.WhenAll` respectively; so it seems that in practice there are few differences between the asynchronous programming models provided by these languages and our `Fetch` monad. However, we believe the differences are important:

- In the asynchronous programming models, concurrency is achieved using special-purpose operations, whereas in our approach existing standard idioms like `sequence` and `mapM` become concurrent automatically by virtue of the `Applicative`

[5] `http://clojure.github.io/core.async/`

instance that we are using. Programmers don't need to learn a new concurrency library; they just use data-fetching operations together with the tools they already know for structuring code, and concurrency comes for free.

- Our system has a built-in cache, which is important for modularity, as we described in Section 2.1.

Explicit blocking (as in `await` above) is often shunned in the asynchronous programming models; instead it is recommended to attach callback methods to the results, like this (in Scala):

```
val future = getData();
future map(x => x + 1);
```

This has the advantage that we don't have to block on the result of the future in order to operate on it, which allows the system to exploit more concurrency. However, the programming style is somewhat indirect; in our system, this would be written

```
do x <- getData; return (x+1)
```

Reactive programming models [8] add another dimension to asynchronous programming, where instead of a single result being returned, there is a stream of results. This is a separate problem space from the one we are addressing in this paper.

11. Further work

The method for taking advantage of concurrency described in Section 4 is fairly simplistic: we run as much computation as possible, and then perform all the data fetching concurrently, repeating these two steps as many times as necessary to complete the computation. There are two ways we could overlap the computation phase with the data-fetching phase:

- As soon as we have the result of any data fetch, we can start running the corresponding blocked part(s) of the computation.
- We might want to emit some requests before we have finished exploring the whole computation. This potentially reduces concurrency but might also reduce latency.

We intend to investigate these in future work.

Acknowledgements

We would like to thank Richard Eisenberg for providing helpful feedback on an early draft of this paper.

References

[1] S. Awodey. *Category Theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, 2006.

[2] K. Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, May 1999.

[3] M. Eriksen. Your server as a function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 5:1–5:7. ACM, 2013. ISBN 978-1-4503-2460-1.

[4] W. L. Harrison. Cheap (but functional) threads. Submitted for publication, `http://people.cs.missouri.edu/~harrisonwl/drafts/CheapThreads.pdf`.

[5] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 189–199, 2007.

[6] S. Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 96–106, 2006.

[7] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008. ISSN 0956-7968.

[8] E. Meijer. Reactive extensions (Rx): Curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 11:1–11:1. ACM, 2010.

[9] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61. ACM, 2006.

[10] E. Scholz. A concurrency monad based on constructor primitives, or, being first-class is not enough. Technical report, Universität Berlin, 1995.

[11] T. Stein, E. Chen, and K. Mangla. Facebook immune system. In *Proceedings of the 4th Workshop on Social Network Systems*, SNS '11, pages 8:1–8:8. ACM, 2011.

[12] W. Swierstra and T. Altenkirch. Beauty in the beast. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 25–36, 2007. ISBN 978-1-59593-674-5.

[13] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin Heidelberg, 2011.

[14] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 791–792, 2012.

[15] J. Vouillon. Lwt: A cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 3–12. ACM, 2008.