

facebook

Haskell in the datacentre

Simon Marlow
March 2017

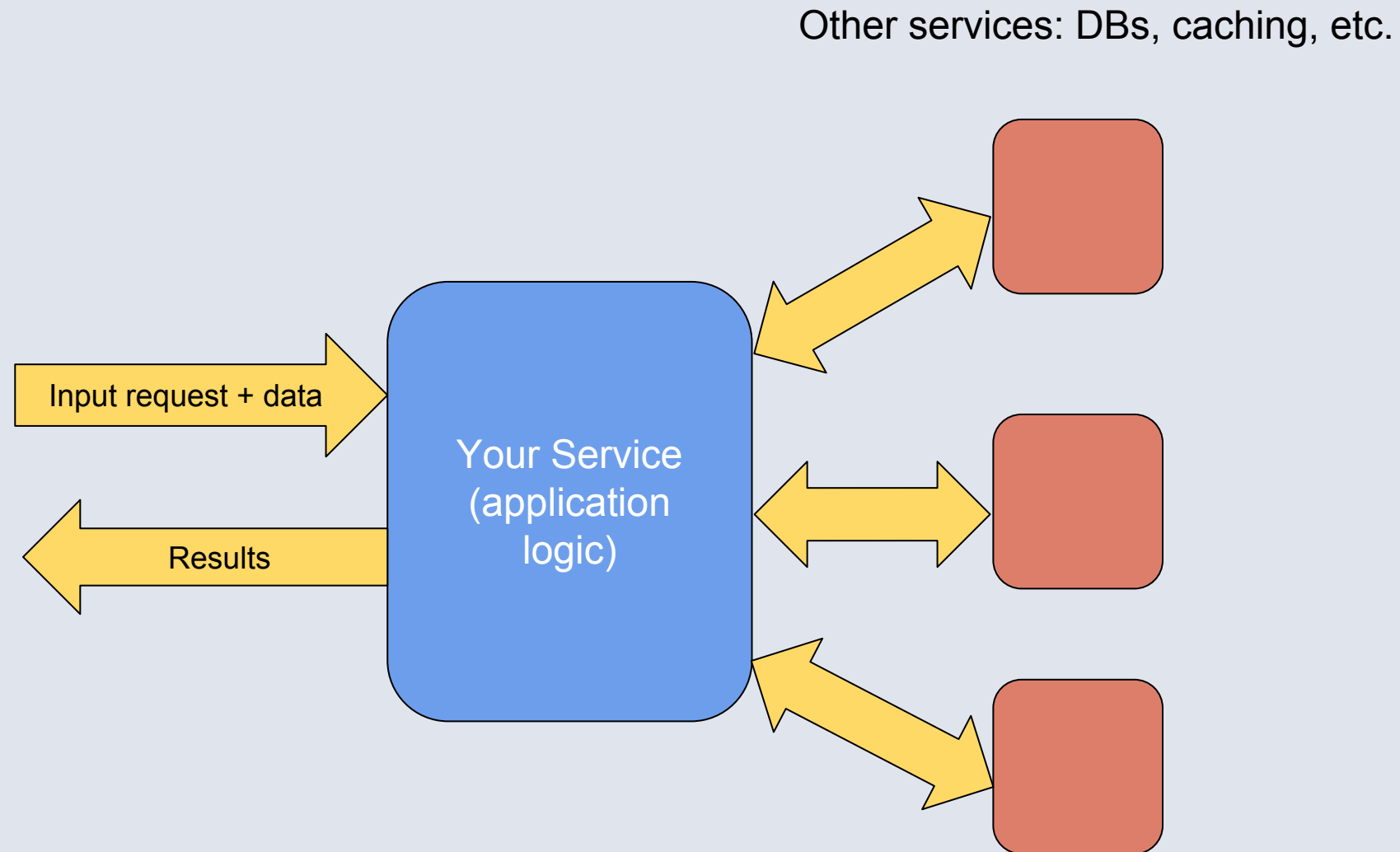




This lecture

- Part 1
 - The Haxl library: making concurrency easy
- Part 2
 - The ApplicativeDo transformation
 - Running Haskell at scale

Our scenario



e.g. web application (result = HTML), other application-logic platforms.

Fetching data efficiently
is the first-order concern.

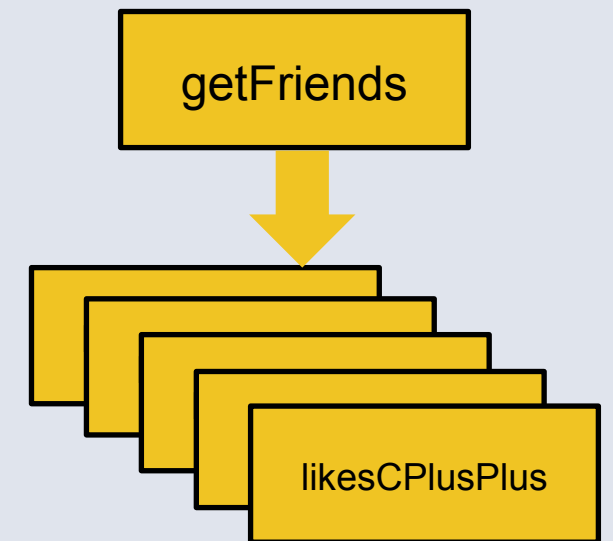
We need concurrency

- Typically spend more time waiting for data than computing
- Multiple independent data-fetch requests must be executed concurrently and/or batched
- Traditional languages and frameworks make the programmer deal with this
 - threads, futures/promises, async, callbacks, etc.
 - Hard to get right, hard to refactor later
 - Application logic mixed with concurrency
 - Clutters the code
 - Clash of concerns

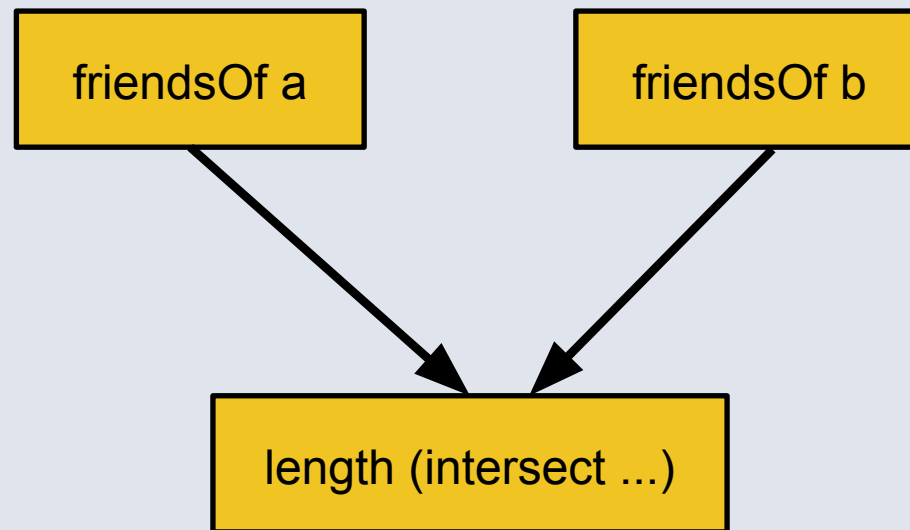
The goal

- (Ignoring side effects for now, assume all data-fetching is effect-free)
- data-fetching is a function
- “just write functional code”
- the framework exploits concurrency as far as data dependencies allow
- The programmer doesn't need to think about it

```
friendsWhoLikeCPlusPlus = do
  friends <- getFriends
  filterM likesCPlusPlus friends
where
  likesCPlusPlus = ...
```




```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```



Haxl: a library for implicit concurrency

- `Hax1` is a Monad
- The implementation of `(>>=)` will allow the computation to block, waiting for data.

```
data Result a
  = Done a
  | Blocked [BlockedRequest] (Hax1 a)

newtype Hax1 a = Hax1 { unHax1 :: IO (Result a) }
```

- Haxl is a Monad
- The implementation of ($>>=$) will allow the computation to block, waiting for data.

```
data Result a
  = Done a
  | Blocked [BlockedRequest] (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

This is the
result of a
computation

- Haxl is a Monad
- The implementation of ($>>=$) will allow the computation to block, waiting for data.

```
data Result a
  = Done a
  | Blocked [BlockedRequest] (Haxl a)
```

Done indicates
that we have
finished

```
newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

- Haxl is a Monad
- The implementation of ($>>=$) will allow the computation to block, waiting for data.

```
data Result a
  = Done a
  | Blocked [BlockedRequest] (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

Blocked indicates that the computation requires this data.

- Haxl is a Monad
- The implementation of ($>>=$) will allow the computation to block, waiting for data.

```
data Result a
  = Done a
  | Blocked [BlockedRequest] (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

This is the continuation to compute the result after the data has been fetched

- Haxl is a Monad
- The implementation of ($>>=$) will allow the computation to block, waiting for data.

```
data Result a
  = Done a
  | Blocked [BlockedRequest] (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

Haxl may need
to do IO

Monad instance

```
instance Monad Haxl where
```

```
  return a = Haxl $ return (Done a)
```

```
  Haxl m >>= k = Haxl $ do
```

```
    r <- m
```

```
    case r of
```

```
      Done a      -> unHaxl (k a)
```

```
      Blocked br c -> return (Blocked br (c >>= k))
```

Monad instance

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

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

Monad instance

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

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

If m blocks with continuation c ,
the continuation for $m \gg= k$ is $c \gg= k$

This is called a *concurrency monad*

- The essence of computations that can pause and continue
- It needs a *scheduler*
- The scheduler will
 - Run the computation
 - If it returns **Done** *a*, we're finished, result is *a*
 - If it returns **Blocked** *reqs cont*:
 - fetch the required data *reqs*
 - continue by executing *cont*

The scheduler

- Assume we have a way to fetch data:

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

- Now, to run a Haxl computation to completion:

```
runHaxl :: Haxl a -> IO a
runHaxl (Haxl h) = do
  result <- h
  case result of
    Done a -> return a
    Blocked br cont -> do
      fetch br
      runHaxl cont
```

But how do we get the result of a data fetch?

- It is possible to plumb it around, but messy and inefficient
- In Haxl we use an IORef:

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

A request for a result of type a

Put the result here (initially Nothing)

- The job of fetch is to fetch the data and fill the IORefs:

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

We need a request operation

```
dataFetch :: Request a -> Haxl a
dataFetch req = Haxl $ do
  ref <- newIORef Nothing
  return (Blocked [BlockedFetch req ref] (get ref))
where
  get ref = Haxl $ do
    Just x <- readIORef ref
    return x
```

Empty ref to
put the result

Continuation:
read the result
from the ref

- (this is simplified; in the real Haxl, Request is a type class so that you can use it with arbitrary data sources)

Example of a Request

Use a GADT:

```
data Request a where
  FriendsOf :: Id -> Request [Id]

friendsOf :: Id -> Haxl [Id]
friendsOf x = dataFetch (FriendsOf x)
```


Yes, but the point was to be able to do *multiple* data-fetches concurrently!

So far you've only told us how to do *one* data-fetch at a time.

- Our example will block on the first friendsOf request:

```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```

blocks here

- How do we allow the Monad to collect multiple data-fetches, so we can execute them concurrently?


First, rewrite to use *Applicative* operators

```
numCommonFriends a b =  
  length <$> (intersect <$> friendsOf a <*> friendsOf b)
```

- When we use *Applicative*, Haxl can collect multiple data fetches and execute them concurrently.
- How?

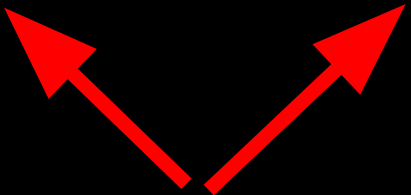
Applicative allows parallelism

$(\gg=) :: \text{Monad } m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$



dependency

$(\langle * \rangle) :: \text{Applicative } f \Rightarrow f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$



independent

Applicative

- You can use `<*>` only when the two computations have no data dependency (they may have effect dependencies though...)
- The implementation of `<*>` is therefore free to exploit this independence for parallelism
- This is why Applicative is a weaker abstraction than Monad: it doesn't provide for data dependency between effects

Applicative instance

```
instance Applicative Haxl where
  pure = return

Haxl f <*> Haxl x = Haxl $ 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))
```

<*> allows both arguments to block waiting for data

<*> can be nested, letting us collect an arbitrary number of data fetches to execute concurrently

Example

```
(intersect <$> friendsOf x) <*> friendsOf y  
=  
(friendsOf x >>= return . intersect) <*> friendsOf y
```

```
(<$>) = fmap  
fmap f m = m >>= return . f
```

Example

```
(friendsOf x >>= return . intersect) <*> friendsOf y  
=  
(Blocked [BlockedRequest (FriendsOf x) rx] (get rx)  
  >>= return . intersect) <*> friendsOf y
```

```
friendsOf x = dataFetch (FriendsOf x)
```

```
dataFetch :: Request a -> Haxl a  
dataFetch req = Haxl $ do  
  ref <- newIORef Nothing  
  return (Blocked [BlockedFetch req ref] (get ref))
```


Example

```
((Blocked [BlockedRequest (FriendsOf x) rx] (get rx)
  >>= return . intersect) <*> friendsOf y
=
(Blocked [BlockedRequest (FriendsOf x) rx]
  (get rx >>= return . intersect)) <*> friendsOf y
```

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

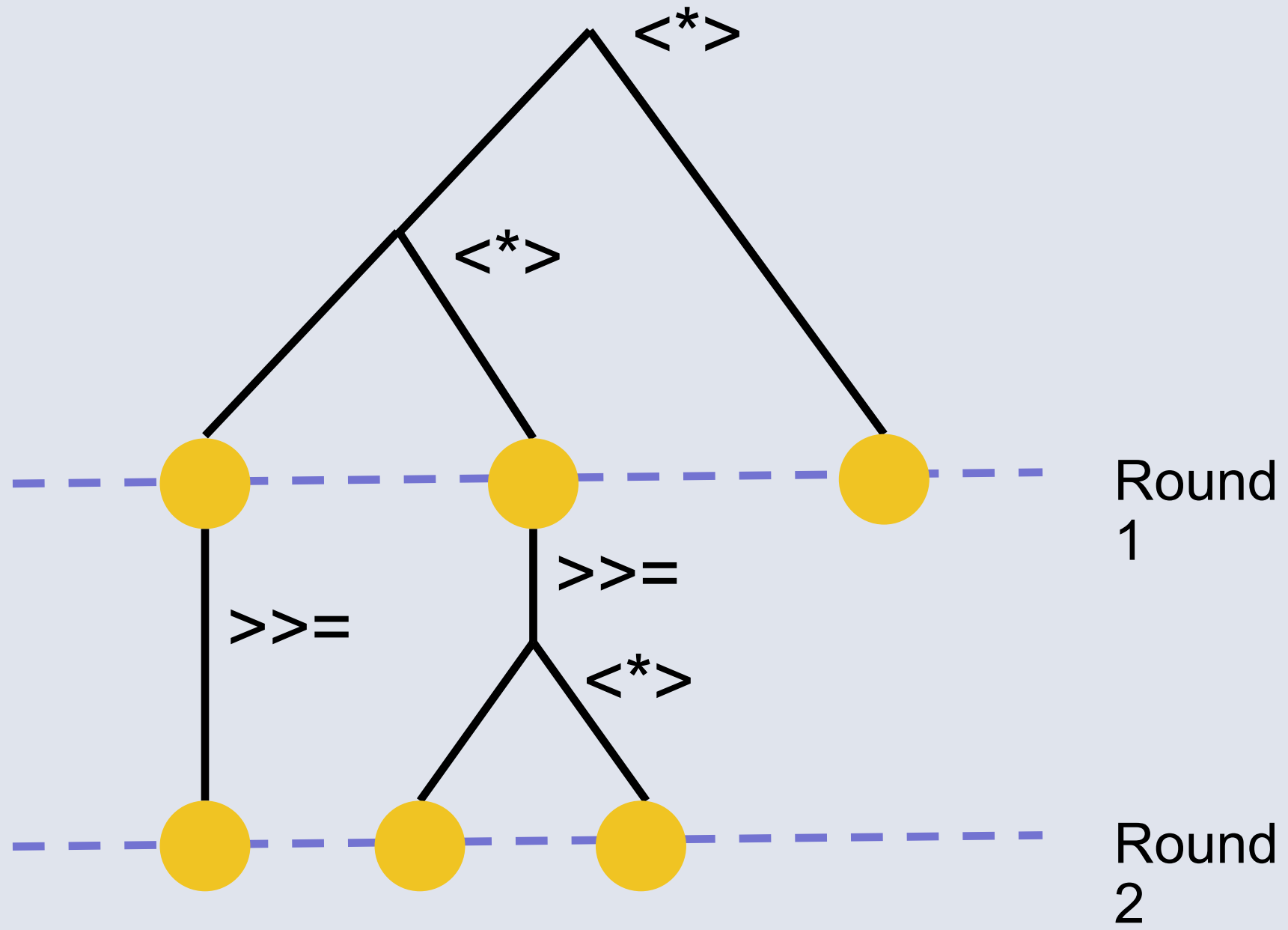
Example

```
(Blocked [BlockedRequest (FriendsOf x) rx]
  (get rx >>= return . intersect)) <*> friendsOf y
=
(Blocked [BlockedRequest (FriendsOf x) rx]
  (get rx >>= return . intersect)) <*>
  Blocked [BlockedRequest (FriendsOf y) ry] (get ry)
```

Example

```
(Blocked [BlockedRequest (FriendsOf x) rx]
  (get rx >>= return . intersect)) <*>
  Blocked [BlockedRequest (FriendsOf y) ry] (get ry)
=
Blocked [ BlockedRequest (FriendsOf x) rx
  , BlockedRequest (FriendsOf y) ry]
  ((get rx >>= return . intersect) <*> get ry)
```

```
Haxl f <*> Haxl x = Haxl $ do
  f' <- f
  x' <- x
  case (f',x') of
    ...
    (Blocked br1 c, Blocked br2 d) ->
      return (Blocked (br1 <> br2) (c <*> d))
```



(Some) Concurrency for free

- Applicative is a standard class in Haskell
- Lots of library functions are already defined using it
- These work concurrently when used with Haxl
- e.g.

```
sequence :: Monad m => [m a] -> m [a]
mapM     :: Monad m => (a -> m b) -> m [a] -> m [b]
filterM  :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

```
friendsLikeCPlusPlus = do
  friends <- getFriends
  cppFriends <- filterM likesCPlusPlus friends
  ...
```

Haxl is a general solution

- ... to the problem of scheduling I/O
- it's useful anywhere that needs to do I/O and doesn't want to express concurrency explicitly.
- Let's write a blog engine.

Concrete example: a blog

- Main pane: posts
- Left pane:
 - Top-10 most popular posts
 - Post topics, with post counts



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

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

renderPage      :: Html -> Html -> Html
renderSidePane  :: Html -> Html -> Html
```



```
data PostId      -- identifies a post
data PostContent -- the content of a post
```

```
-- metadata about a post
data PostInfo = PostInfo
  { postId      :: PostId
  , postDate    :: Date
  , ...
  }
```

```
-- data-fetching operations
getPostIds      :: Haxl [PostId]
getPostInfo     :: PostId -> Haxl PostInfo
getPostContent  :: PostId -> Haxl PostContent
```

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

```
mainPane :: Haxl Html
```

```
mainPane = do
```

```
posts <- getAllPostsInfo
```

```
let ordered = take 5 $ sortBy (flip (comparing postDate)) posts
```

```
content <- mapM (getPostContent . postId) ordered
```

```
return $ renderPosts (zip ordered content)
```

First we fetch
all the
metadata

Sort it by date,
and take the
latest 5

Fetch the
content for
those 5

And finally
render the
output

Things to note

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

- No explicit concurrency constructs
 - Just standard structuring tools: do-notation, <*>, mapM
 - No concurrency bugs

No manual batching

```
getAllPostsInfo :: Hax1 [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

This line
performs many
data fetches

Unbatched

```
SELECT postinfo FROM posts
  WHERE postid = id1
```

```
SELECT postinfo FROM posts
  WHERE postid = id2
```

...

Batched

```
SELECT postinfo FROM posts
  WHERE postid IN {id1, id2, ...}
```

- Multiple parts of our application might want to access the same data
 - e.g. in the blog, we query the list of posts in multiple panes

```
mainPane :: Haxl Html
mainPane = do
  posts <- getAllPostsInfo
  ...
```

```
topics :: Haxl Html
topics = do
  posts <- getAllPostsInfo
  ...
```

Caching

- For efficiency, fetch the data only once
- But factoring out the fetching logic and passing the results around would be
 - anti-modular
 - unnecessary code churn and clutter
- Therefore: Haxl ***caches*** all data fetches automatically
- Multiple identical requests result in just one remote fetch

- Ubiquitous caching is liberating: don't worry about duplicating work, just write the logic you want.

Taking caching a step further

- At the end of the computation, the cache contains all the data we fetched
- If we run the computation again,
 - it will use the cached data
 - and return exactly the same result as before, guaranteed
- We can exploit this!
 - Run the code & save the cache, we now have a unit test that works even when the real data changes
 - We can debug errors that happen in production

- Haxl provides `dumpCacheAsHaskell`
 - (demo)
 - copy/paste result to make a unit test
 - at Facebook we have a custom GHCi command to create tests

Taking caching even further...

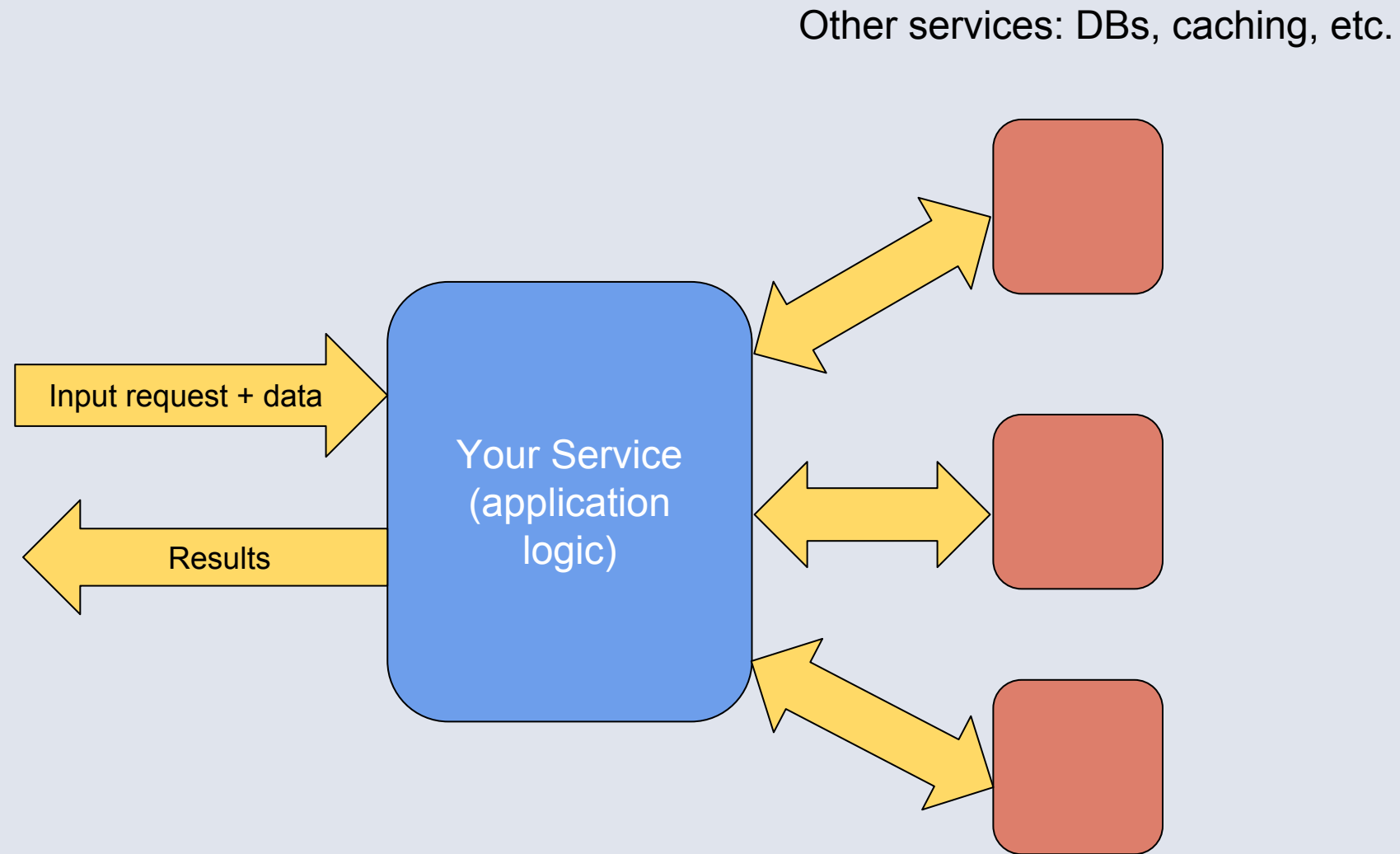
- We can cache not just data-fetches, but the results of arbitrary computations.
- This is called *memoization*
- Haxl supports memoization via an operation:

```
cachedComputation :: Key -> Haxl a -> Haxl a
```

- Memoization is liberating, just like caching:
 - we can stop worrying about duplicating work.
 - share code, not work.
 - Increases modularity, no need to plumb results around.
 - Just write the logic you want.
 - (needs profiling to decide where to memoize)

End of Part 1!

Our scenario



e.g. web application (result = HTML), other application-logic platforms.

Recap

- Fetching data efficiently is the first-order concern
 - (we typically spend more time waiting for data than computing)
- Getting concurrency right by hand is
 - difficult
 - annoying
 - fragile
 - distracts from the logic you're trying to implement

```
numCommonFriends :: Id → Id → Hax1 Int
numCommonFriends x y =
  length <$>
    (intersect
      <$> friendsOf x
      <*> friendsOf y)
```

Parallel data-fetch

```
[FriendsOf x, FriendsOf y]
```

But that's not enough

- Writing in Applicative style by hand can be painful
- The programmer has to spot where they can use `<*>`

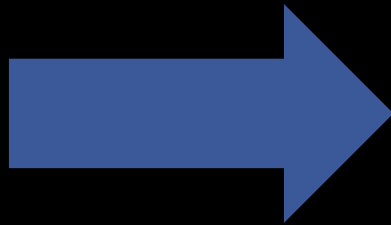
```
numCommonFriends :: Id → Id → Haxl Int
numCommonFriends x y = do
  fx ← friendsOf x
  fy ← friendsOf y
  return (length (intersect fx fy))
```

Sequential data-fetches

```
[FriendsOf x]
[FriendsOf y]
```

Getting it right in more complex cases can be really hard...


```
do x1 ← a
  x2 ← b x1
  x3 ← c
  x4 ← d x3
  x5 ← e x1 x4
  return (x2,x4,x5)
```



```
do ((x1,x2),x4) ← (,)
  <$> (do x1 ← a
        x2 ← b x1
        return (x1,x2))
  <*> (do x3 ← c; d x3)
  x5 ← e x1 x4
  return (x2,x4,x5)
```

What shall we do instead?

- Let the user write **do**-notation
- Have the compiler translate to `<*>` where possible
- (as a special case, we also get to use **do**-notation for types that are `Applicative` but not `Monad`.)

Desugaring Haskell's `do`-notation Into Applicative Operations

Simon Marlow

Facebook
smarlow@fb.com

Simon Peyton Jones

Microsoft Research
simonpj@microsoft.com

Edward Kmett

McGraw Hill Financial
ekmett@mhfi.com

Andrey Mokhov

Newcastle University
andrey.mokhov@ncl.ac.uk

Abstract

Monads have taken the world by storm, and are supported by `do`-notation (at least in Haskell). Programmers are increasingly waking up to the usefulness and ubiquity of `Applicative`s, but they have so far been hampered by the absence of supporting notation. In this paper we show how to re-use the very same `do`-notation to work for `Applicative`s as well, providing efficiency benefits for some types that are both `Monad` and `Applicative`, and syntactic convenience for those that are merely `Applicative`. The result is fully implemented in GHC, and is in use at Facebook to make it easy to write highly-parallel queries in a distributed system.

1. Introduction

Consider this Haskell function that calculates the number of common friends between two Facebook users:

```
numCommonFriends :: Id → Id → Haxl Int
numCommonFriends x y = do
  fx ← friendsOf x
  fy ← friendsOf y
  return (length (intersect fx fy))
```

Here `friendsOf` is an operation that makes a remote query to a database to fetch the list of friends of a user. Desugaring the monadic `do` expression according to the Haskell standard [10]

a `Monad` lies an `Applicative` [13]. To be concrete, we can rewrite `numCommonFriends` using `Applicative` combinators like this:

```
numCommonFriends :: Id → Id → Haxl Int
numCommonFriends x y =
  (λfx fy → length (intersect fx fy))
  <$> friendsOf x
  <*> friendsOf y
```

The combinators `<$>` and `<*>` are defined in Figure 1, but for now we simply note that the two calls to `friendsOf` are now manifestly independent of one another. And indeed the implementation of the `Haxl` `Monad` can take advantage of that independence to perform the two `friendsOf` queries in parallel; in fact it collects them together and batches them into a single query.

But there is still a problem; programmers should not have to spot where they can use `<*>` to gain its advantages, because they are likely to miss some opportunities, especially when code is refactored. Moreover there are maintainability and comprehensibility benefits in using a single universal notation, namely `do` notation. In this paper we show how to have our cake and eat it too: the programmer writes `do` notation, and the compiler desugars it automatically into the efficient parallel code that uses `Applicative` combinators. We make these contributions:

- Rather than desugaring `do` notation uniformly into `Monad` combinators, we show how to take advantage of the program's dependency structure to selectively use `Applicative` combinators.

Why is this useful?

- Just write a sequence of statements
- Compiler analyses the dependencies and extracts the maximum parallelism by transforming the sequence using `<*>` where possible
- We don't have to think about dependencies
- We cannot miss any opportunities accidentally

Start with a simple example

```
do x1 <- A
   x2 <- B
   return (x1,x2)
```

Standard
Haskell
desugaring

```
A >>= \x1 ->
B >>= \x2 ->
return (x1,x2)
```

ApplicativeDo

```
(,) <$> A <*> B
```

equivalent

```
f <$> ma = ma >>= \a ->
           return (f a)
```

```
mf <*> ma = mf >>= \f ->
           ma >>= \a ->
           return (f a)
```

Dependencies prevent `<*>`

```
do x1 <- A
   x2 <- B[x1]
   return (x1,x2)
```

- Now we cannot use `<*>`, because B depends on x1
- This is the essence of the difference between Applicative and Monad:

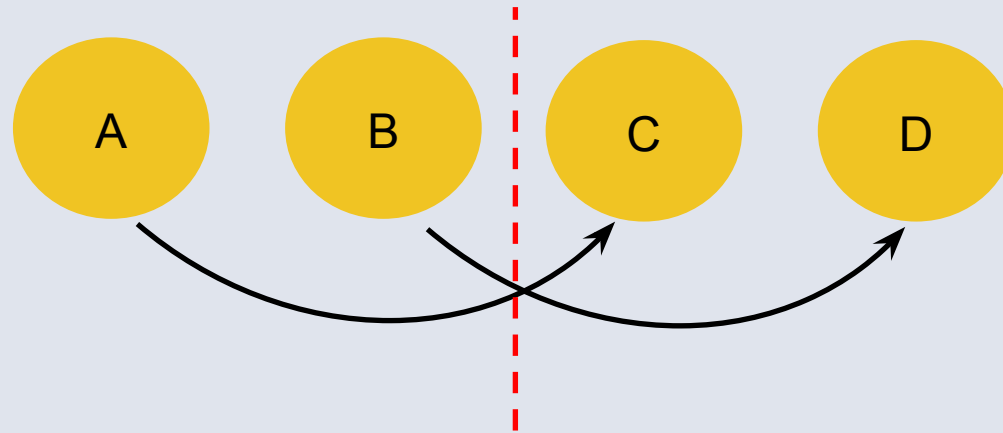
```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(>>=) :: Monad f      => f a -> (a -> f b) -> f b
```

- So we want to use Applicative when *possible* but Monad when *necessary*.

Mixing it up

- What about

```
do x1 <- A  
  x2 <- B  
  x3 <- C[x1]  
  x4 <- D[x2]  
  return (x3, x4)
```



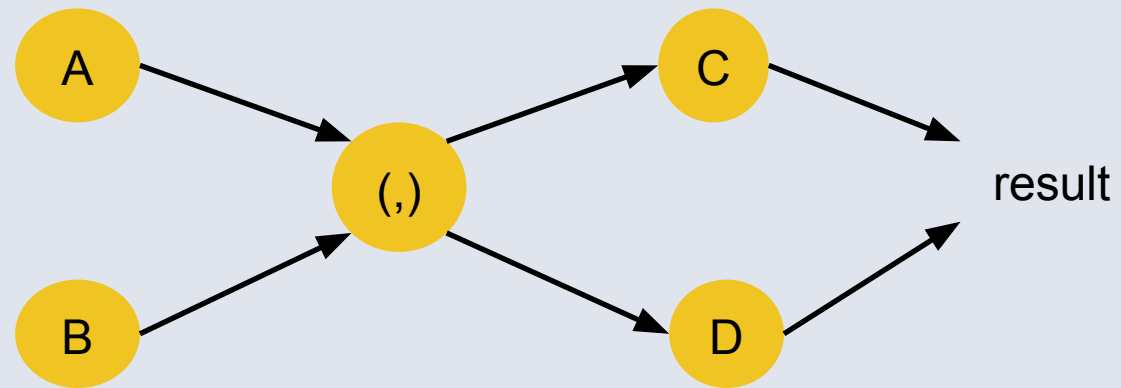
```
do x1 <- A
   x2 <- B
   x3 <- C[x1]
   x4 <- D[x2]
   return (x3,x4)
```

ApplicativeDo



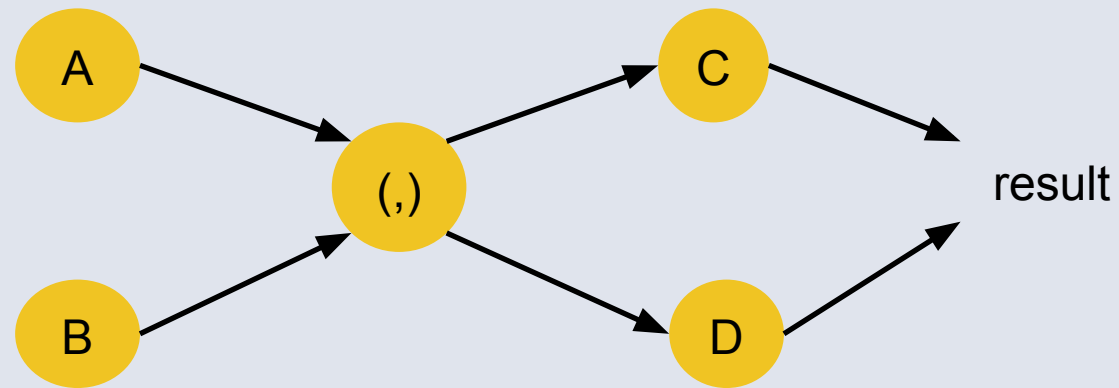
```
((,) <$> A <*> B) >>= \(x1,x2) ->
(,) <$> C[x1] <*> D[x2]
```


But is that the best translation?



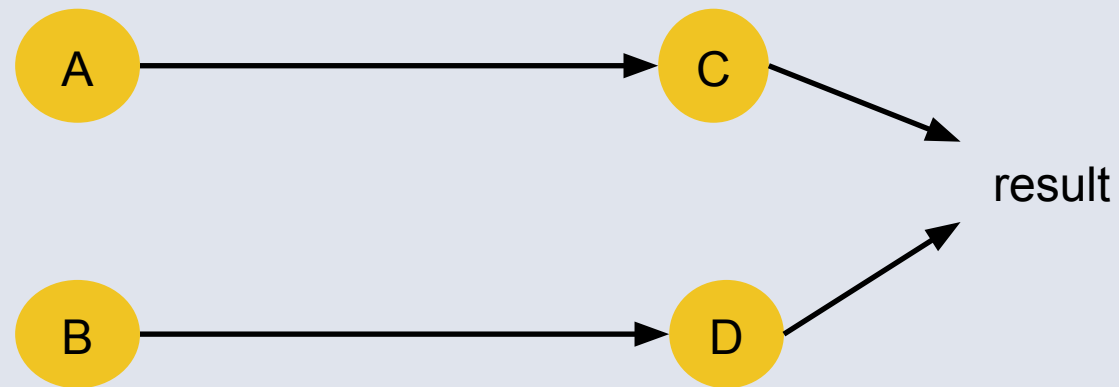
$(A \mid B) ; (C \mid D)$

But is that the best translation?



$(A \mid B) ; (C \mid D)$

- But we only have dependencies $A \rightarrow C$ and $B \rightarrow D$, so why not



$(A ; C) \mid (B ; D)$

Evaluating cost

- Take a simple parallel cost model
 - “|” = “max”
 - “;” = “+”
- e.g. take $A = 2, B = 1, C = 1, D = 2$

$(A | B) ; (C | D)$ cost: 4

$(A ; C) | (B ; D)$ cost: 3

Alternative translations

$(A \mid B) ; (C \mid D)$

```
((, ) <$> A <*> B) >>= \(x1, x2) ->  
(, ) <$> C[x1] <*> D[x2]
```

$(A ; C) \mid (B ; D)$

```
(, ) <$> (A >>= \(x1 -> C[x1])  
      <*> (B >>= \(x2 -> D[x2])
```

But...

```
do x1 <- A
   x2 <- B
   x3 <- C[x1]
   x4 <- D[x2]
   return (x3,x4)
```



NO!

```
(, ) <$> (A >>= \x1 -> C[x1])
      <*> (B >>= \x2 -> D[x2])
```

- This is not semantically equivalent to the original
- Effects would take place in the order A,C,B,D

But do we really care about ordering?

- Haxl doesn't – or at least, there are no effects to observe
- But we do want exceptions to be deterministic:

```
do x1 <- A
   x2 <- B
   x3 <- C[x1]
   x4 <- D[x2]
   return (x3,x4)
```

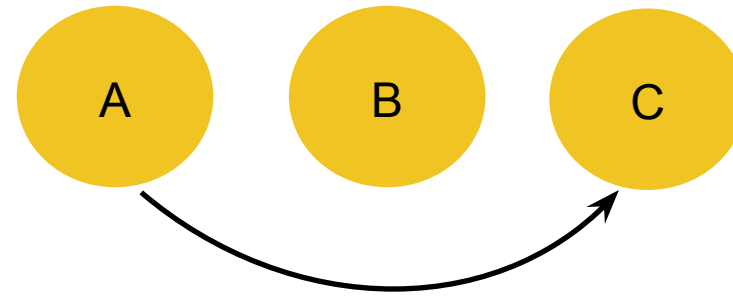
- If B and C throw exceptions, I always want B's exception.
- Reordering to A,C,B,D would break this.

Preserving equivalence is good

- It means `ApplicativeDo` works with any `Monad/Applicative` that satisfies the laws.
- If we reordered things, it would only work on commutative Monads.

What does optimal mean?

```
do x <- A  
  y <- B  
  z <- C[x]  
  return (y + z)
```



Choices:

	A=0, B=2, C=1	A=1, B=2, C=0
(A B) ; C	3	2
A ; (B C)	2	3

- We don't know the costs at compile time.
- Therefore, be conservative.
- Our goal:

Choose a translation that is optimal when all statements have equal cost.

- (there may be multiple valid solutions)

(Aside: use “join”)

```
do x1 <- A
   x2 <- B
   x3 <- C[x1]
   x4 <- D[x2]
   return (x3,x4)
```

ApplicativeDo



```
((,) <$> A <*> B) >>= \ (x1,x2) ->
(,) <$> C[x1] <*> D[x2]
```

• Better:

```
join :: Monad m => m (m a) -> m a
join m = m >>= id
```

```
join ((\x1 x2 -> (,) C[x1] <*> D[x2]) <$> A <*> B)
```

Algorithm sketch

- Two stages:

```
do x1 <- A  
   x2 <- B[x1]  
   x3 <- C  
   return (x2, x3)
```

rearrangement

```
{ x1 <- A; x2 <- B[x1] } | { x3 <- C }
```

desugaring

```
(\x2 x3 -> (x2, x3))  
  <$> (A >>= \x1 -> B[x1])  
  <*> C
```

Rearrangement

- Start with a list of statements

$$L = \{ s_1 ; \dots ; s_n \}$$

- Introduce “parallel blocks”

$$s = (L_1 \mid \dots \mid L_n)$$

- Meaning: just flatten the list

- A parallel block will turn into an applicative expression

```
do x1 <- A
   x2 <- B[x1]
   x3 <- C
   return (x2, x3)
```



rearrangement

$$\{ x1 <- A; x2 <- B[x1] \} \mid \{ x3 <- C \}$$

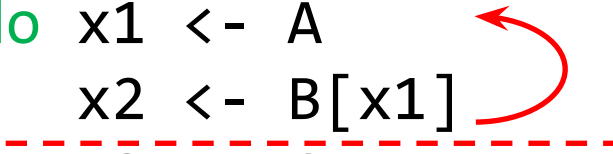
Where do we introduce parallel blocks?

- Take the sequence without the final return
- (desugaring will put it back later)

```
do x1 <- A
   x2 <- B[x1]
   x3 <- C
   return (x2, x3)
```

- Split the sequence into *segments*
- Place a segment boundary between two statements when there are no dependencies that cross the boundary
- Make a parallel block from the segments; apply recursively

```
do x1 <- A
   x2 <- B[x1]
   x3 <- C
   return (x2, x3)
```



```
rearrange { x1 <- A; x2 <- B[x1] }
| rearrange { x3 <- C }
```

What if there are no segments?

- If it's a single statement: we're done
- Otherwise we need a ";" somewhere
- In this case we have no choice:

```
rearrange { x3 <- C }  
= { x3 <- C }
```

```
rearrange { x1 <- A; x2 <- B[x1] }  
= { x1 <- A; x2 <- B[x1] }
```

- (we'll do a more complex example shortly)
- Result of rearrangement:

```
{ x1 <- A; x2 <- B[x1] } | { x3 <- C }
```

Next, desugar to get an expression

```
desugar ({ x1 <- A; x2 <- B[x1] } | { x3 <- C }) (x2,x3)
```

The expression
from “return”

- desugaring a parallel block yields an Applicative expression:

```
(\x2 x3 -> (x2,x3))  
  <$> desugar { x1 <- A; x2 <- B[x1] } x2  
  <*> desugar { x3 <- C } x3
```

```
(\x2 x3 -> (x2,x3))
  <$> desugar { x1 <- A; x2 <- B[x1] } x2
  <*> desugar { x3 <- C } x3
```

- First, deal with this:

```
desugar { x3 <- C } x3
  =
  C
```

- Next:

```
desugar { x1 <- A; x2 <- B[x1] } x2
  =
  A >>= \x1 -> desugar { x2 <- B[x1] } x2
  =
  A >>= \x1 -> B[x1]
```


Result

```
do x1 <- A  
   x2 <- B[x1]  
   x3 <- C  
   return (x2,x3)
```



```
(\x2 x3 -> (x2,x3))  
  <$> (A >>= \x1 -> B[x1])  
  <*> C
```

A more complex example

```
x1 <- A
x2 <- B[x1]
x3 <- C
x4 <- D[x3]
x5 <- E[x1, x4]
return (x2, x4, x5)
```

- Rearrange:
 - There are no segments
 - We have to insert “;” somewhere
 - And end up with the optimal result

Finding the optimal result

- Just evaluate all possibilities:

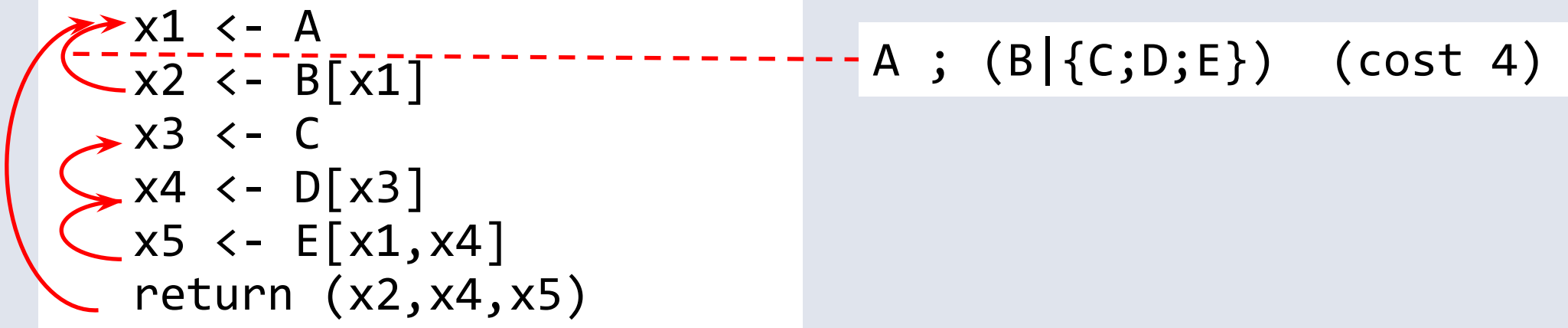
- Starting with $\{ s_1 ; \dots ; s_n \}$

- For each i in $2..n$, compute

- rearrange $\{ s_1 ; \dots ; s_{i-1} \}$; rearrange $\{ s_i ; \dots ; s_n \}$

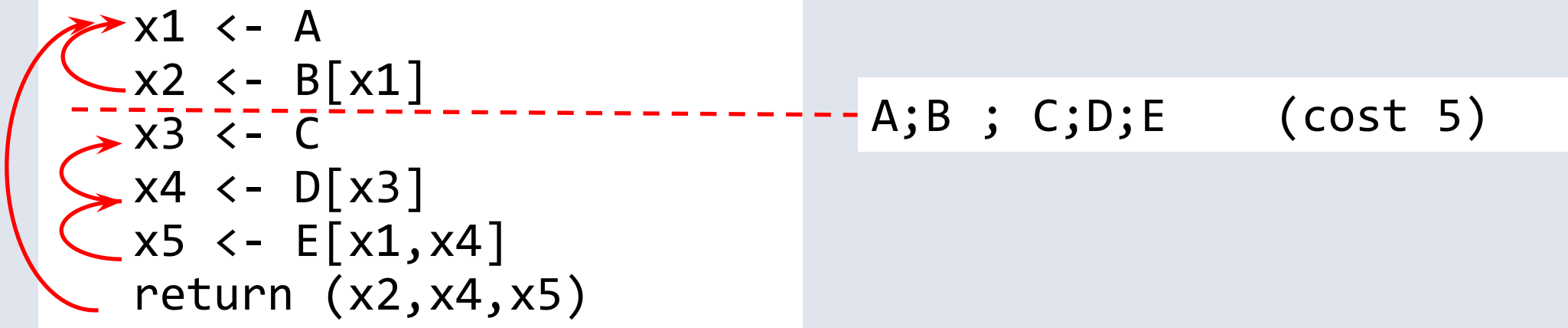
- Evaluate with parallel cost model, with “|” = “max” and “;” = “+”
 - Every statement costs 1
 - Pick the cheapest!

```
x1 <- A  
x2 <- B[x1]  
x3 <- C  
x4 <- D[x3]  
x5 <- E[x1, x4]  
return (x2, x4, x5)
```



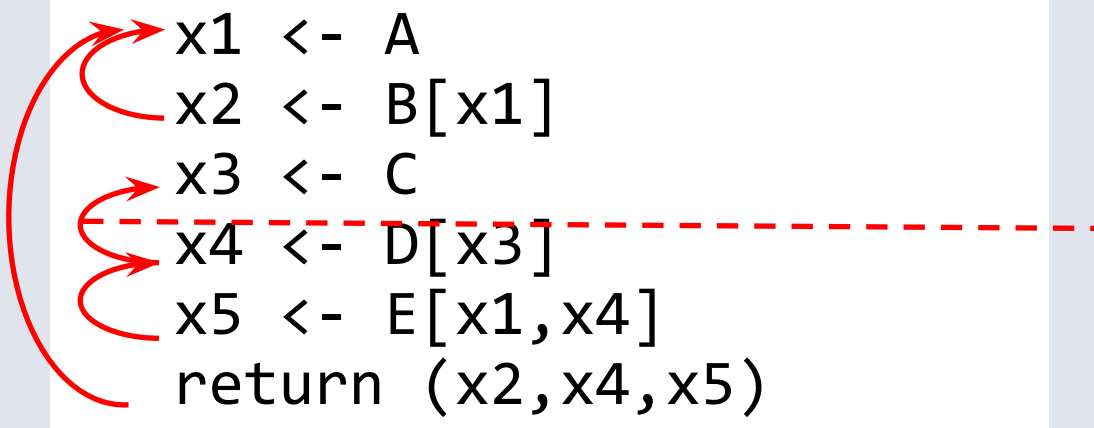
```
A ; (B|{C;D;E}) (cost 4)
```

```
x1 <- A
x2 <- B[x1]
x3 <- C
x4 <- D[x3]
x5 <- E[x1, x4]
return (x2, x4, x5)
```



```
A;B ; C;D;E (cost 5)
```

```
x1 <- A
x2 <- B[x1]
x3 <- C
x4 <- D[x3]
x5 <- E[x1, x4]
return (x2, x4, x5)
```



```
({A;B}|C) ; D;E (cost 4)
```

```
x1 <- A
x2 <- B[x1]
x3 <- C
x4 <- D[x3]
x5 <- E[x1, x4]
return (x2, x4, x5)
```

$(\{A;B\} | \{C;D\}) ; E$ (cost 3)

We have a winner!

- After desugaring:

```
join (\(x1,x2) x4 ->
      E[x1,x4] >>= \x5 -> pure (x2,x4,x5))
<$> (A >>= \x1 -> B[x1] >>= \x2 -> return (x1,x2))
<*> (C >>= \x3 -> D[x3])
```

- Full details in the paper, “Translating Haskell’s do-notation into Applicative operations” (Haskell Symposium 2016)

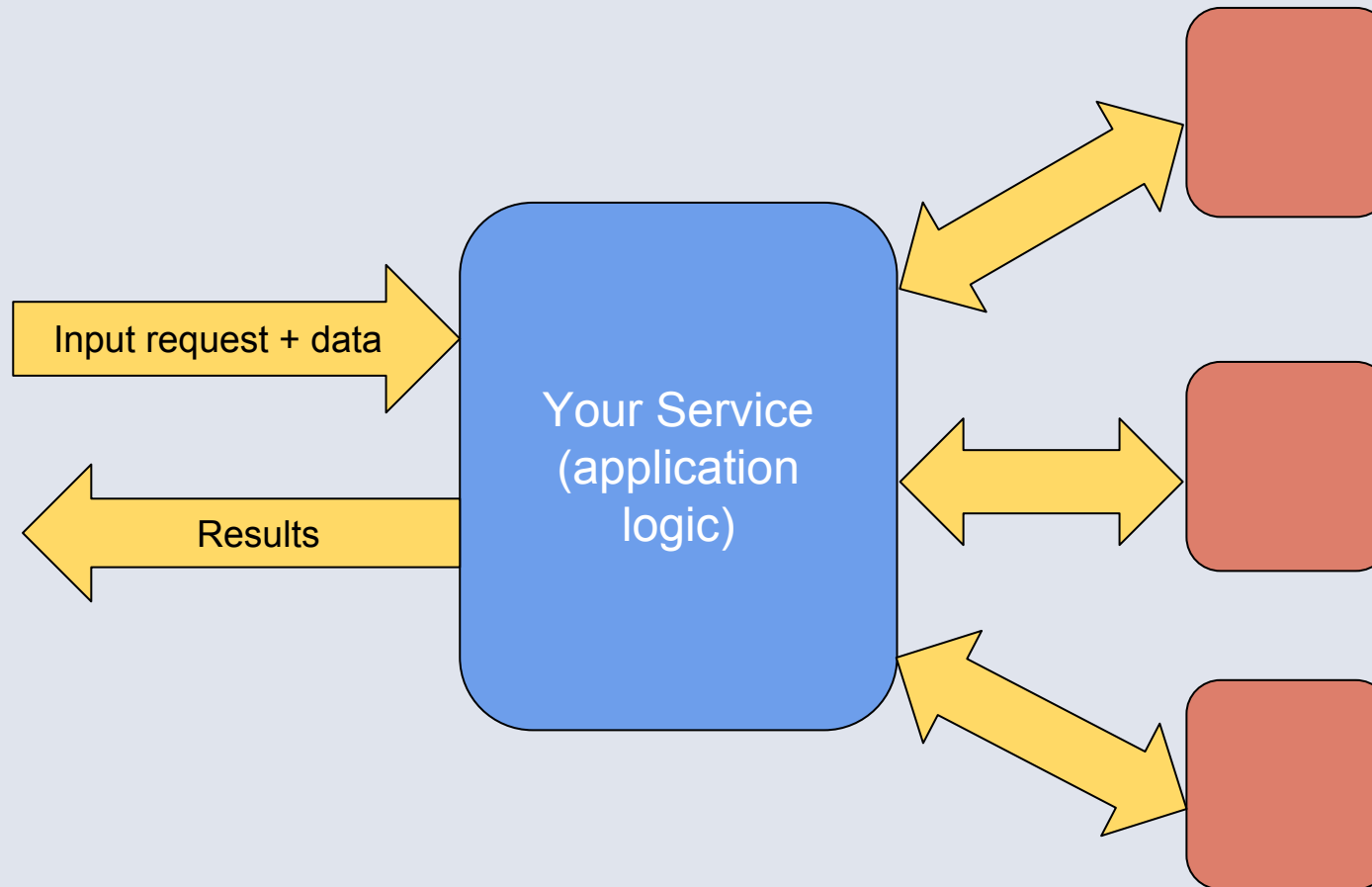
Results

- This transform is being used across our codebase at Facebook
 - (hundreds of thousands of lines of code)
- Users typically don't worry about concurrency
- ApplicativeDo gives 20-50% improvement in request latency
 - (there is already concurrency from mapM, explicit <*>, etc.)

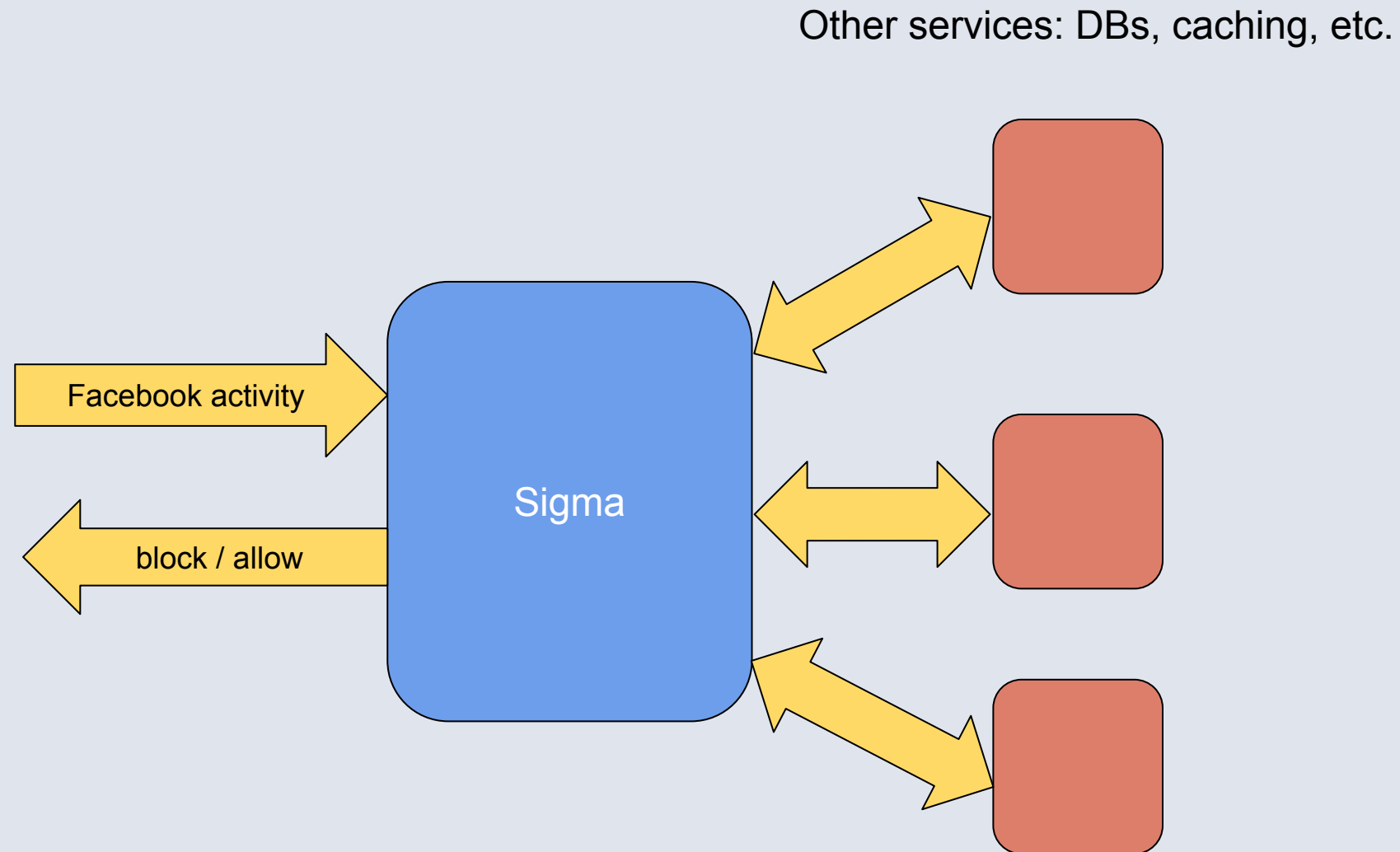
Haskell in the datacentre

Our scenario

Other services: DBs, caching, etc.



Our scenario



- Sigma is a *rule engine* used to detect abusive activity
- Rule logic is written in Haskell, using Haxl + ApplicativeDo

Stats

- Haskell-based Sigma has been running now for over 2 years
- Serves over 1M requests per second
- Largest Haskell deployment in the world:
 - thousands of machines across 6 datacentres
- Handles abuse detection for multiple different teams & products
- Many dozens of people writing Haskell day to day

Moving fast

- “git push” pushes to production(!)
 - Code changes are running live within minutes of pushing
 - But not *too* fast:
 - code review
 - build failure prevents push
 - automated unit tests
 - automated profiling runs to detect regressions

What kind of Haskell do we write?

- Custom Prelude
 - Haxl stuff + common APIs
 - Partial functions removed (head, undefined...)
 - Due to difficulty of debugging when they happen in production
- Mainly Tasty + HUnit for unit tests
 - (QuickCheck in a few places)
- HLint to enforce style and catch obvious opportunities for cleanup
 - also to prevent use of deprecated APIs
- Automated tools for refactoring across the whole codebase
 - great for API migrations, cleaning up after deprecations

Tooling

- Custom GHCi “haxlsh”
 - Used for all development & debugging
 - Links in external C++ libraries to talk to data sources
 - Means we can interactively develop and test code against production data
 - “Remote GHCi”
 - Interpreted code runs in profiling mode, so we get stack traces for exceptions
-

Tooling cont.

- Our own Thrift to Haskell compiler
 - Thrift is a language-independent RPC protocol used in FB
 - Thrift spec defines types and requests for a service
 - Our compiler uses GADTs and DataKinds to statically enforce compiler correctness
 - For most external data, we talk to the service using Haskell code generated by our Thrift compiler
 - Lightweight Haskell threads scale better than C++ threads

Performance

- We did a few things to improve performance:
 - Tweaks to the GC
 - Instead of fixed-size per-CPU nurseries, a pool of nursery chunks
 - Instead of one worker thread per core, N worker threads per core but use fewer for GC
 - Improved load balancing
 - NUMA-optimised memory allocation
 - Worker threads pinned to NUMA nodes
 - Nurseries are pinned to the local NUMA node

facebook

Questions?