Introduction to Arrows

Incrementalization

```
foo :: (Integer, Bool) -> String
foo (a, b) =
  let c = f a ← really expensive!
      d = g c b
  in h c d
```

Idea: cache result of `f a` if `a` does not change.

Incrementalization

How can we implement caching in Haskell?

Simplest answer: take the previous values as an argument.

```
foo :: (Integer, Char) -> (Integer, Bool) -> (String, Char)
foo (a_prev, c_prev) (a, b) =
  let c | a == a_prev = c_prev
        | otherwise   = f a
      d = g c b
  in (h c d, c)
```

Incrementalization

This approach sucks. :(

1. Leaks implementation details
2. Hard to read
3. Doesn’t scale
4. Easy to screw up

We want an abstraction!

Abstracting incrementalization

Caching monad?

```
class Monad m => MonadCache m where
  cache :: m a -> m a
```

```
foo :: MonadCache m => (Integer, Bool) -> m String
foo (a, b) = do
  c <- cache $ f a
  d <- g c b
  h c d
```

But wait: how does `cache` know what changed?

Abstracting incrementalization

```
cache :: MonadCache m => m a -> m a
```

```
cache :: (MonadCache m, Eq a) => (a -> m b) -> a -> m b?
```

```
cache f a
```

```
cache (\() -> f a) ()
```

Abstracting incrementalization

```
g a b c = do
  x <- someFunc a b
  y <- anotherFunc b c
  z <- thirdFunc x y
  fourthFunc c z
```

Abstracting incrementalization

```
g a b c = do
  x <- someFunc a b
  y <- anotherFunc b c
  flip cache (x, y) $ \(x', y') -> do
    z <- thirdFunc x' y'
    fourthFunc c z
```

Abstracting incrementalization

`Monad` is too powerful.

Can we get away with just `Applicative`?

Answer: not really.

We need an abstraction *between* `Applicative` and `Monad`.

Arrows

Arrows

What is an arrow?

`Monad m`	`m a`	A value `a`, plus some context in `m`.

`Arrow arr`	`arr a b`
	``(a `arr` b)``
		A function `a -> b`,
		plus some context in `arr`.

Arrows

```
class Monad m => MonadCache m where
  cache :: m a -> m a
```

```
class Arrow arr => ArrowCache arr where
  cache :: Eq a => (a `arr` b) -> (a `arr` b)
```

Arrows

1. How do you create an arrow?

2. How do you run an arrow?

3. How do you compose two arrows together?

Arrows

Lifting

```
pure :: Monad m => a -> m a
arr :: Arrow arr => (a -> b) -> (a `arr` b)
```

Composition

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
(>>>) :: Arrow arr =>
  (a `arr` b) -> (b `arr` c) -> (a `arr` c)
```

Arrows

This interface is very restrictive!

```
m1 :: a -> m Bool     m2 :: a -> m b     m3 :: a -> m b
     f a = m1 a >>= \b -> if b then m2 a else m3 a
```

```
a1 :: a `arr` Bool     a2 :: a `arr` b     a3 :: a `arr` b
                  f a = a1 a >>> ???
```

Arrows

`m >>= f` ← black box!

Monads are *higher-order.*

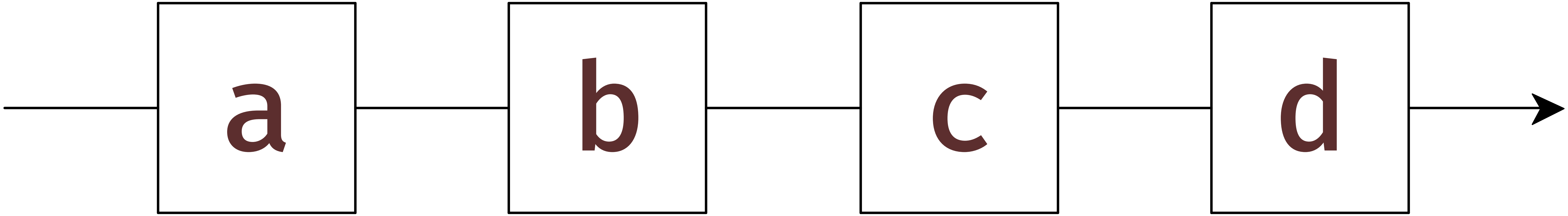
```
join :: Monad m => m (m a) -> m a
```

```
a1 >>> a2 >>> a3 >>> a4
```

Arrows are *first-order.*

Arrows as graphs

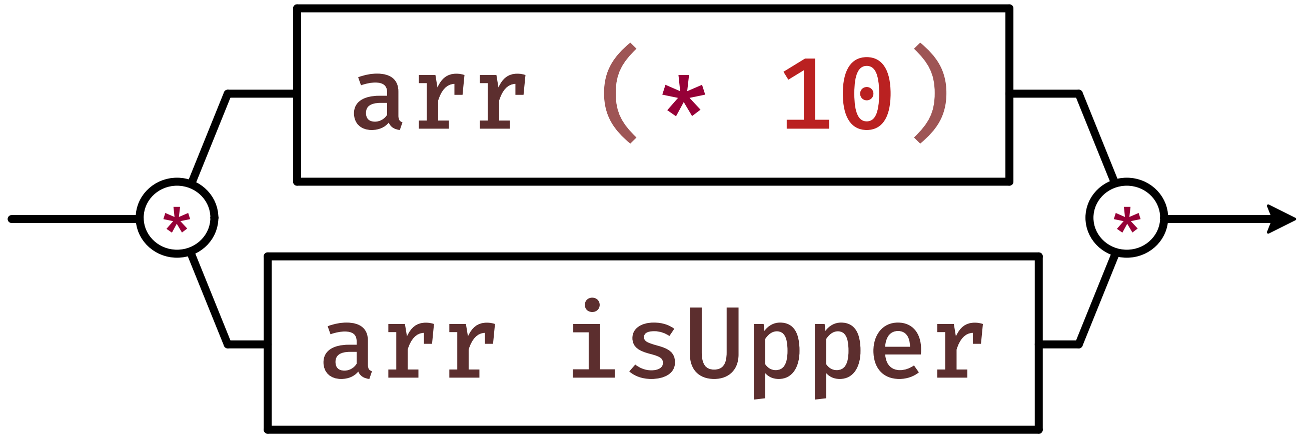
```
a >>> b >>> c >>> d
```



Arrows as graphs: products

```
arr (* 10) :: Integer `arr` Integer
arr isUpper :: Char `arr` Bool
```

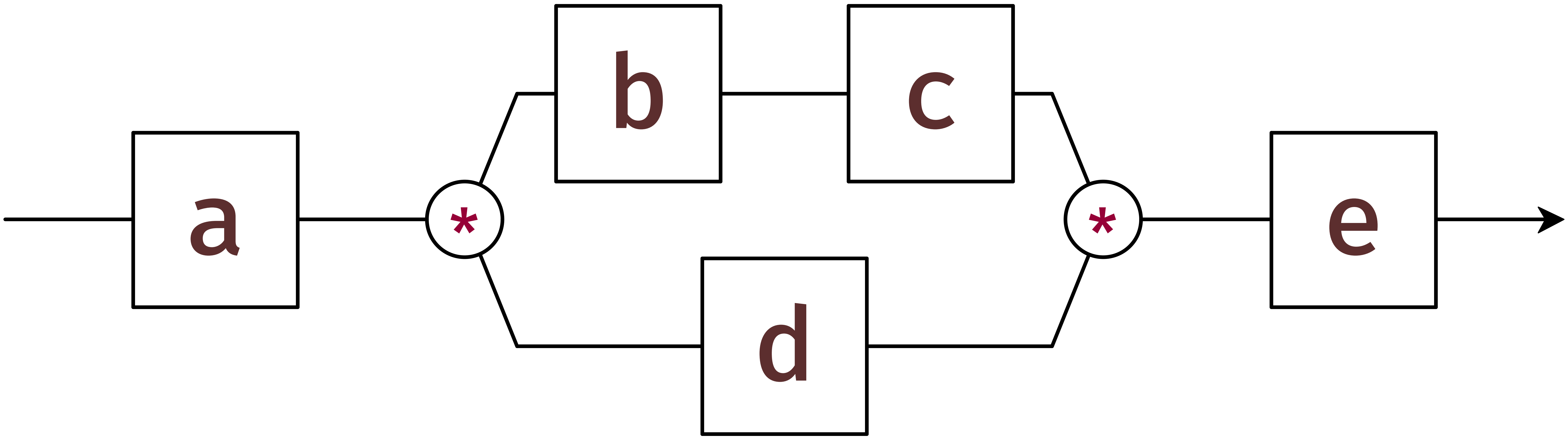
```
(Integer, Char) `arr` (Integer, Bool)
```



```
(***) :: Arrow arr =>
  (a `arr` b) -> (c `arr` d) -> ((a, c) `arr` (b, d))
```

Arrows as graphs: products

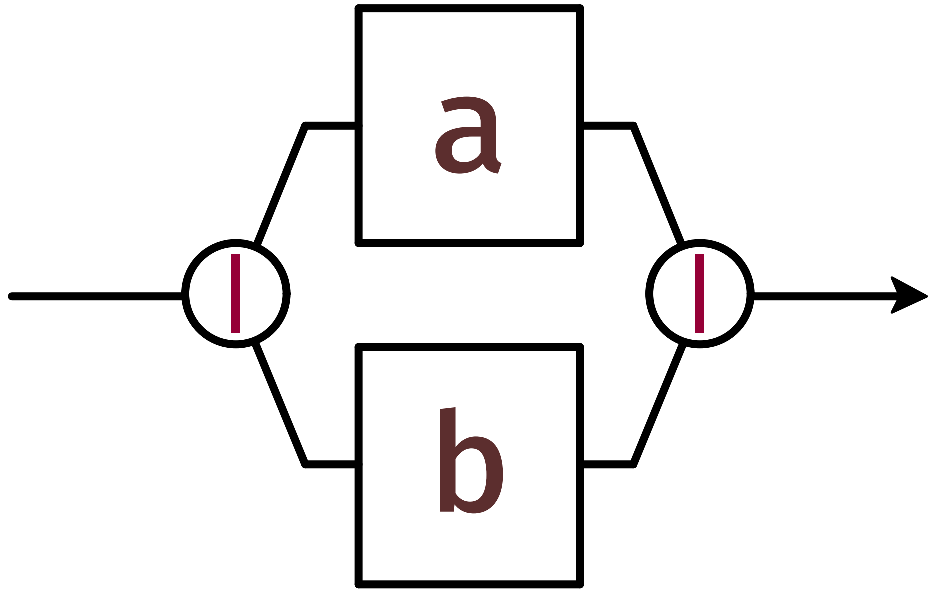
```
a >>> ((b >>> c) *** d) >>> e
```



Arrows as graphs: sums

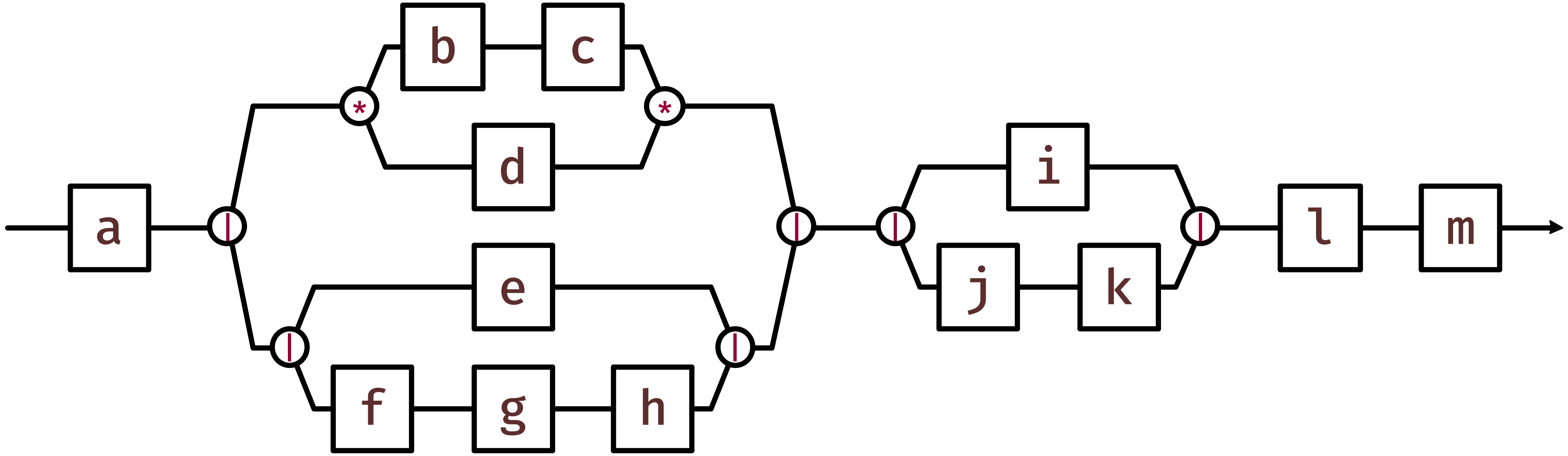
```
class Arrow arr => ArrowChoice arr where
  (|||) :: (a `arr` b) -> (c `arr` d)
        -> (Either a c `arr` Either b d)
```

```
a ||| b
```



Arrows as graphs

```
  arr :: Arrow arr => (a -> b) -> (a `arr` b)
(>>>) :: Arrow arr => (a `arr` b) -> (b `arr` c) -> (a `arr` c)
(***) :: Arrow arr => (a `arr` b) -> (c `arr` d) -> ((a, c) `arr` (b, d))
(|||) :: ArrowChoice arr =>
           (a `arr` b) -> (c `arr` d) -> (Either a c `arr` Either b d)
```



```
a >>> (((b >>> c) *** d) ||| (e ||| (f >>> g >>> h)))
  >>> (i ||| (j >>> k)) >>> l >>> m
```

Arrow notation

```
foo :: (A, B) -> M C
foo = \(a, b) -> do
  x <- f a
  y <- g (b, x)
  h (x, y)
```

```
foo :: (A, B) `Arr` C
foo = proc (a, b) -> do
  x <- f -< a
  y <- g -< (b, x)
  h -< (x, y)
```

Arrow notation

```
do { pat <- expr; ...; expr }

proc pat -> do { pat <- cmd; ...; cmd }

cmd ::= expr -< expr | ...
```

Arrow notation

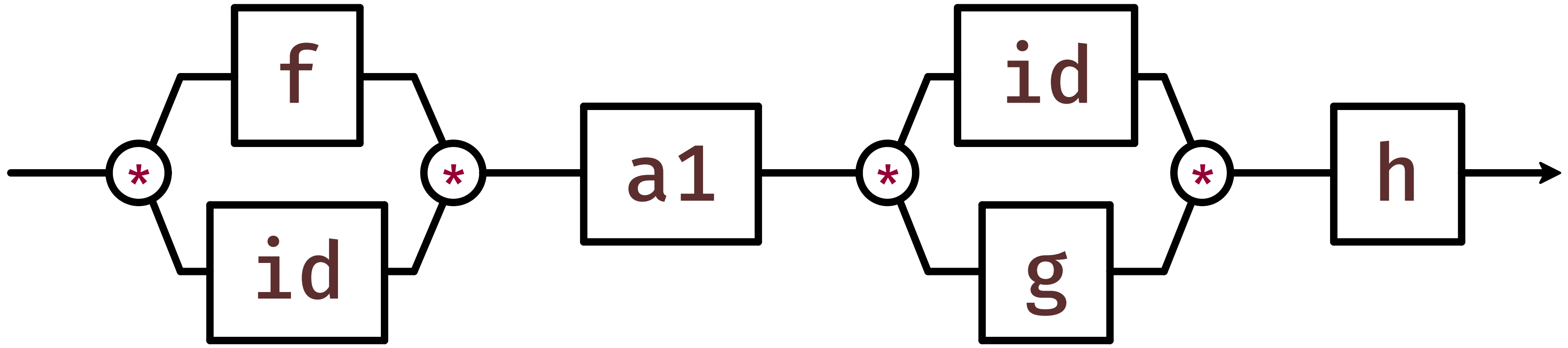
```
foo :: (A, B) `Arr` C
foo = proc (a, b) -> do
    x <- f -< a
    y <- g -< (b, x)
    h -< (x, y)
  where
    f :: A `Arr` D
    f = proc a -> ...
```

Arrow notation

```
foo :: (A, B) `Arr` C
foo = proc (a, b) -> do
  x <- f -< a
  y <- g -< (b, x)
  h -< (x, y)
```

→

```
let a1 = arr (\(x, b) -> (x, (b, x)))
in (f *** id) >>> a1 >>> (id *** g) >>> h
```



Arrow notation

```
cmd ::= expr -< expr
     |  do { pat <- cmd; ...; cmd }
     |  ...
```

```
proc (a, b) -> do
  x <- do
    y <- f -< a
    g -< (y, b)
  h -< x
```

Arrow notation

```
cmd ::= expr -< expr
     |  do { pat <- cmd; ...; cmd }
     |  case expr of { pat -> cmd; ... }
     |  ...
```

```
proc (a, b) -> do
  x <- case b of
    Just c  -> f -< c
    Nothing -> g -< a
  h -< x
```

Arrow notation

```
cmd ::= expr -< expr
     |  do { pat <- cmd; ...; cmd }
     |  case expr of { pat -> cmd; ... }
     |  if expr then cmd else cmd
     |  ...
```

```
proc (a, b) -> do
  x <- if f b
    then g -< a
    else h -< b
  i -< x
```

Arrow notation: control operators

```
f x = do
  y <- g x `catchError` \e -> h e x
  for y $ \z -> do
    ...
```

```
f = proc x -> do
  y <- ???
  ???
```

Arrow notation: control operators

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

```
class Arrow arr => ArrowError e arr | arr -> e where
  throwA :: e `arr` a
  catchA :: (a `arr` b) -> ((a, e) `arr` b) -> (a `arr` b)
```

```
g `catchA` proc (x, e) -> h -< (e, x)
```

Arrow notation: control operators

```
proc (a, b) -> do
  x <- (g `catchA` proc (_, e) -> h -< (b, e)) -< a
  ...
```

Arrow notation: control operators

```
proc (a, b) -> do
  x <- (g' `catchA` h') -< (a, b)
  ...
where
  g' = proc (a, _) -> g -< a
  h' = proc ((_, b), e) -> h -< (b, e)
```

Arrow notation: control operators

```
cmd ::= expr -< expr
     |  do { pat <- cmd; ...; cmd }
     |  case expr of { pat -> cmd; ... }
     |  if expr then cmd else cmd
     |  \pat ... -> cmd
     |  cmd infix_expr cmd
     |  ...
```

Arrow notation: control operators

```
cmd ::= expr -< expr
     |  do { pat <- cmd; ...; cmd }
     |  case expr of { pat -> cmd; ... }
     |  if expr then cmd else cmd
     |  \pat ... -> cmd
     |  cmd infix_expr cmd
     |  ...
```

```
proc (a, b) -> do
  x <- (g -< a) `catchA` \e -> h -< (b, e)
  ...
```

Arrow notation: control operators

```
cmd ::= expr -< expr
     |  do { pat <- cmd; ...; cmd }
     |  case expr of { pat -> cmd; ... }
     |  if expr then cmd else cmd
     |  \pat ... -> cmd
     |  cmd infix_expr cmd
     |  (| expr cmd ... |)
     |  ...
```

```
proc (a, b) -> do
  x <- (g -< a) `catchA` \e -> h -< (b, e)
  ...
```

Arrow notation: control operators

```
cmd ::= expr -< expr
      | do { pat <- cmd; ...; cmd }
      | case expr of { pat -> cmd; ... }
      | if expr then cmd else cmd
      | \pat ... -> cmd
      | cmd infix_expr cmd
      | (| expr cmd ... |)
      | ...
```

```
proc (a, b) -> do
  x <- (| catchA (g -< a) (\e -> h -< (b, e)) |)
  ...
```

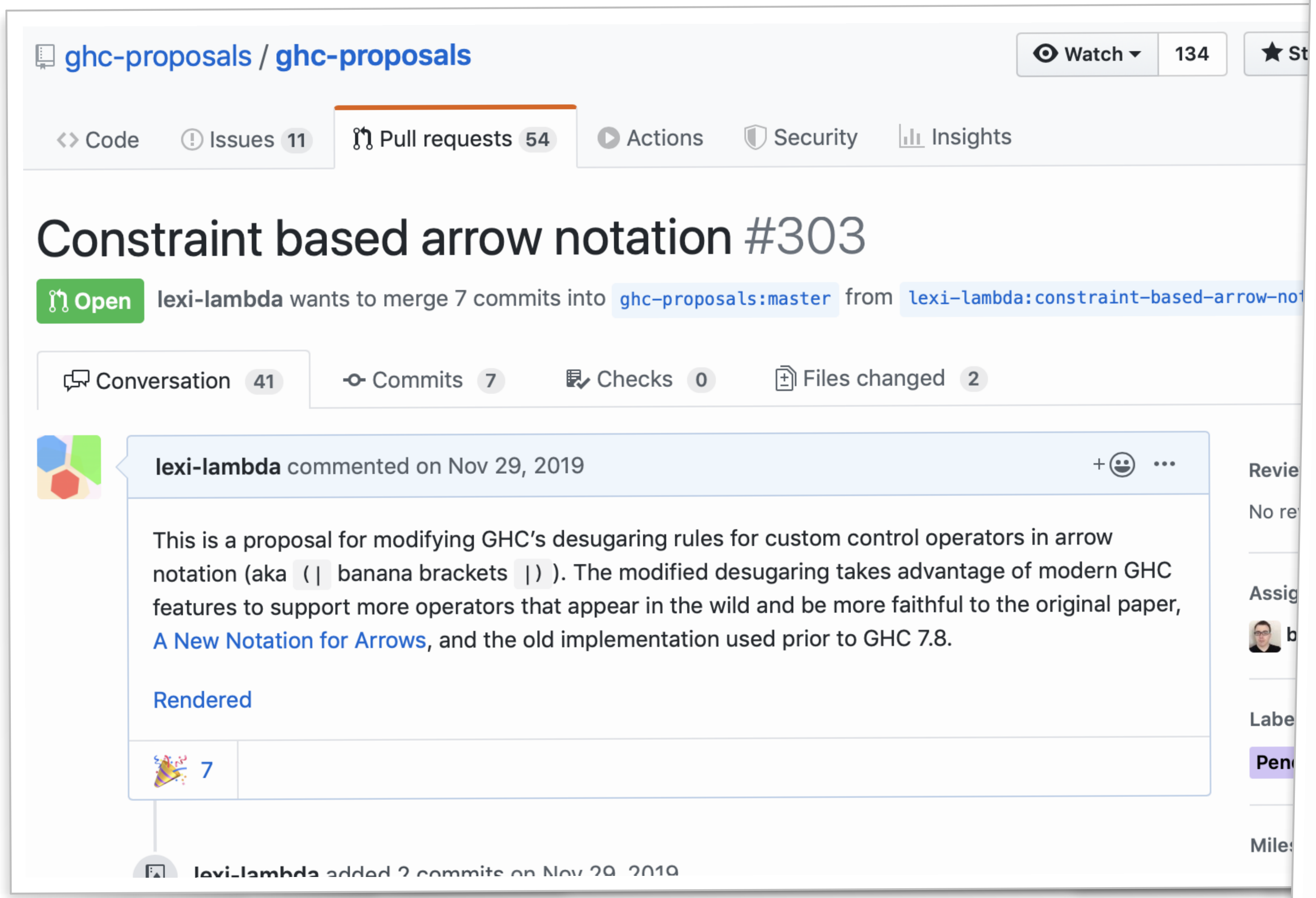
Arrow notation: control operators

```
          catchA :: ArrowError e arr =>
            (a `arr` b) -> ((a, e) `arr` b) -> (a `arr` b)

catchA :: ArrowError e arr =>
  ((a, ()) `arr` b) -> ((a, (e, ())) `arr` b) -> ((a, ()) `arr` b)
```

Arrow notation: control operators

```
{- Note [Weird control operator types]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Arrow notation (i.e. `proc`) has support for so-called “custom control operators,” which allow
things like

    proc (x, y) -> do
      z <- foo -< x
      (f -< z) `catchA` \e -> g -< (y, e)

to magically work. What’s so magical about that? Well, note that `catchA` is an ordinary function,
but it’s being given /commands/ as arguments, not expressions. Also note that the arguments to
`catchA` reference the variables `y` and `z`, which are bound earlier in the `proc` expression as
arrow-local variables.

To make this work, GHC has to thread `y` and `z` through `catchA` in the generated code, which will
end up being something like this:

        arr (\(x, y) -> (x, (x, y)))
    >>> first foo
    >>> arr (\(z, (x, y)) -> (z, y))
    >>> catchA (first f)
               (arr (\((_, y), e) -> (y, e)) >>> g)
```

Arrow notation: control operators



Constraint based arrow notation #303

Open lexi-lambda wants to merge 7 commits into `ghc-proposals:master` from `lexi-lambda:constraint-based-arrow-not`

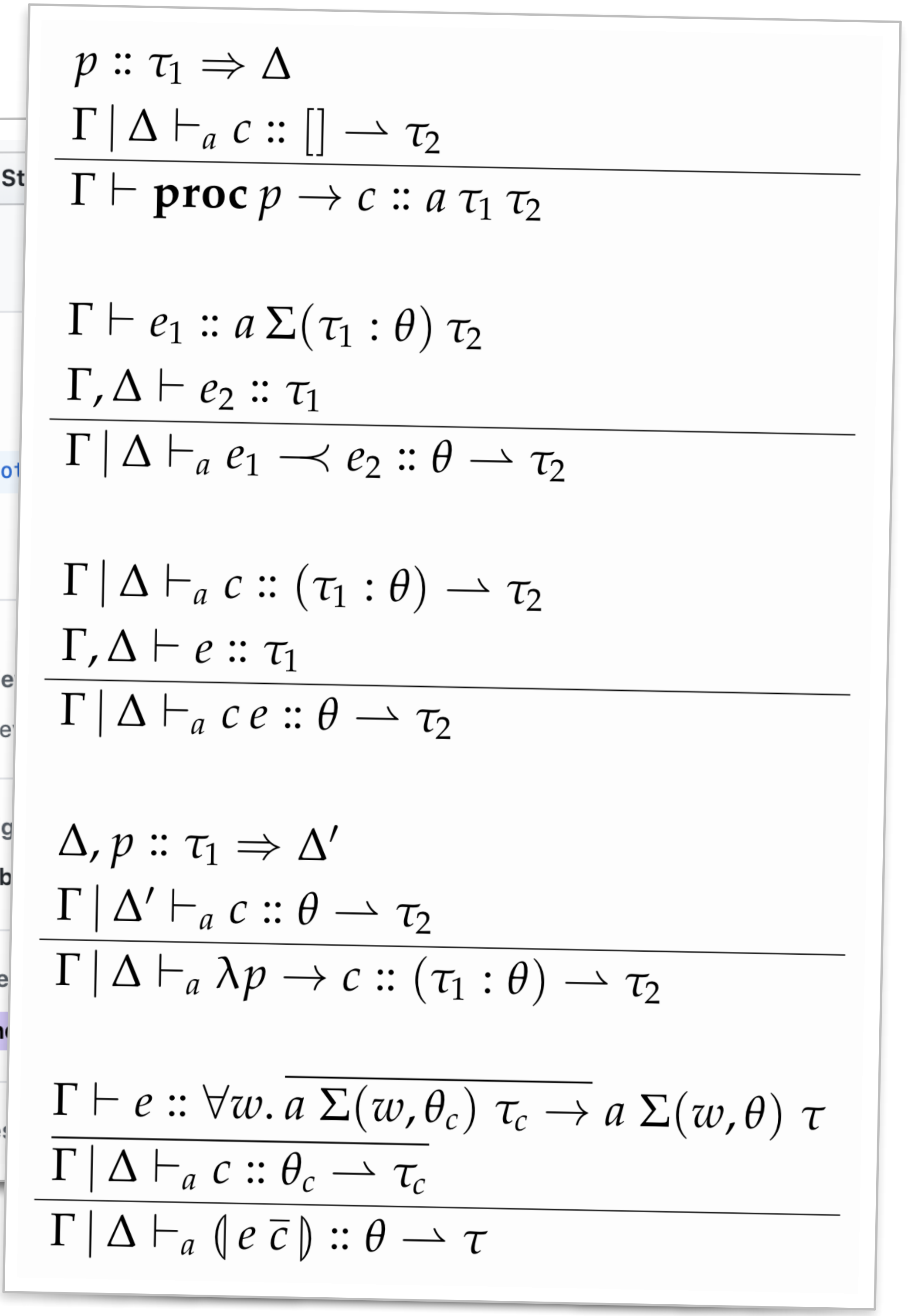
lexi-lambda commented on Nov 29, 2019

This is a proposal for modifying GHC's desugaring rules for custom control operators in arrow notation (aka `(|` banana brackets `|)`). The modified desugaring takes advantage of modern GHC features to support more operators that appear in the wild and be more faithful to the original paper, A New Notation for Arrows, and the old implementation used prior to GHC 7.8.

Rendered

🎉 7

lexi-lambda added 2 commits on Nov 29, 2019



$$\frac{\begin{array}{l} p :: \tau_1 \Rightarrow \Delta \\ \Gamma \mid \Delta \vdash_a c :: [\,] \rightharpoonup \tau_2 \end{array}}{\Gamma \vdash \mathbf{proc}\; p \to c :: a\; \tau_1\; \tau_2}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 :: a\, \Sigma(\tau_1 : \theta)\; \tau_2 \\ \Gamma, \Delta \vdash e_2 :: \tau_1 \end{array}}{\Gamma \mid \Delta \vdash_a e_1 \mathrel{-\!\!\prec} e_2 :: \theta \rightharpoonup \tau_2}$$

$$\frac{\begin{array}{l} \Gamma \mid \Delta \vdash_a c :: (\tau_1 : \theta) \rightharpoonup \tau_2 \\ \Gamma, \Delta \vdash e :: \tau_1 \end{array}}{\Gamma \mid \Delta \vdash_a c\; e :: \theta \rightharpoonup \tau_2}$$

$$\frac{\begin{array}{l} \Delta, p :: \tau_1 \Rightarrow \Delta' \\ \Gamma \mid \Delta' \vdash_a c :: \theta \rightharpoonup \tau_2 \end{array}}{\Gamma \mid \Delta \vdash_a \lambda p \to c :: (\tau_1 : \theta) \rightharpoonup \tau_2}$$

$$\frac{\begin{array}{l} \Gamma \vdash e :: \forall w.\, \overline{a\, \Sigma(w, \theta_c)\; \tau_c} \to a\, \Sigma(w, \theta)\; \tau \\ \overline{\Gamma \mid \Delta \vdash_a c :: \theta_c} \rightharpoonup \tau_c \end{array}}{\Gamma \mid \Delta \vdash_a (\!|\, e\; \overline{c} \,|\!) :: \theta \rightharpoonup \tau}$$

Resources

GHC User’s Guide section on arrow notation

https://downloads.haskell.org/ghc/8.8.1/docs/html/users_guide/glasgow_exts.html#arrow-notation

A New Notation for Arrows

http://www.staff.city.ac.uk/~ross/papers/notation.html

GHC Proposal: Constraint-based arrow notation

https://github.com/ghc-proposals/ghc-proposals/pull/303

Ask me!