This is Driving Me Loopy: Transforming `loop` in Arrowized Functional Reactive Programs

Finnbar Keating

Okay, that title is a lot

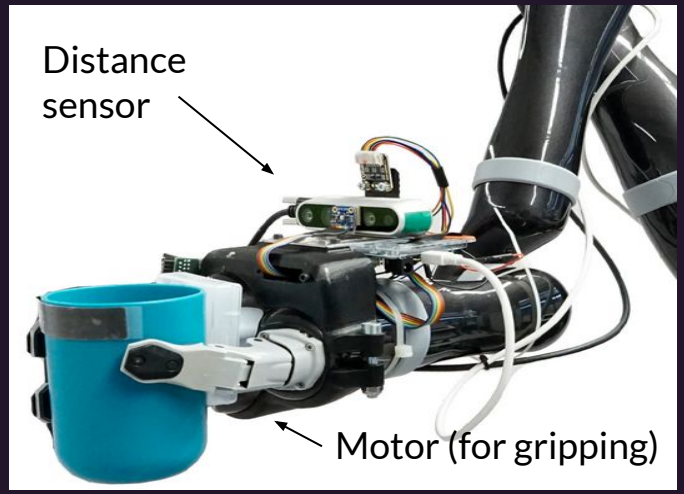
This talk is split into three parts!

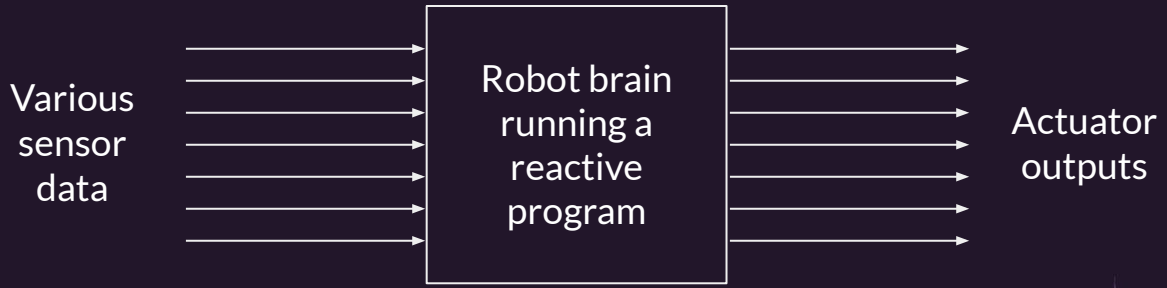
1. What is Arrowized Functional Reactive Programming (AFRP), and what is `loop` within that?
2. Why is it driving me loopy?
3. How do we make it drive me less loopy?

(image reproduced from Visual Sensor Fusion Based Autonomous Robotic System for Assistive Drinking, Sensors ‘21)

Reactive Programs

A reactive program is one which takes in inputs at each time step (a time delta) to produce outputs at the same rate, as to *react* to those inputs.

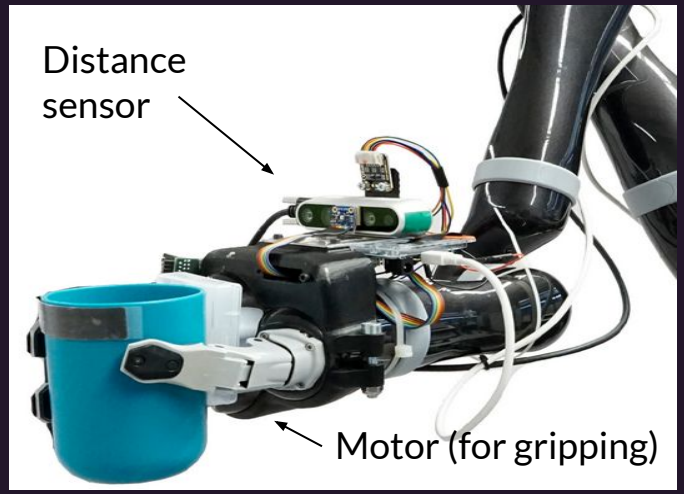


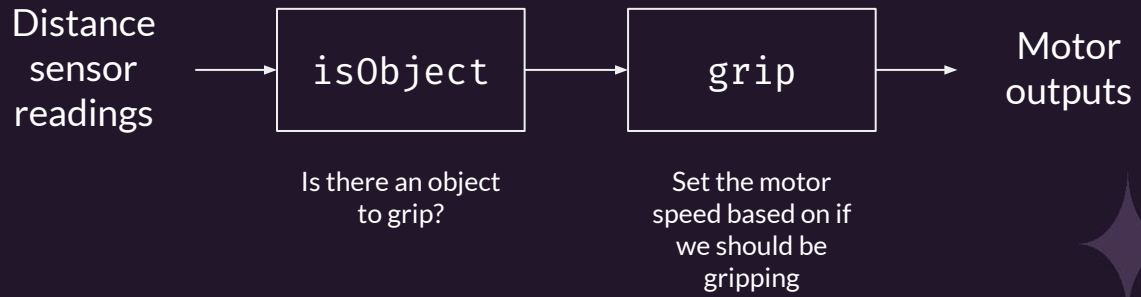


Reactive Programs

A reactive program is one which takes in inputs at each time step (a time delta) to produce outputs at the same rate, as to *react* to those inputs.

We can represent those programs as a *dataflow diagram* to show how the inputs are used at each time step to produce the outputs.



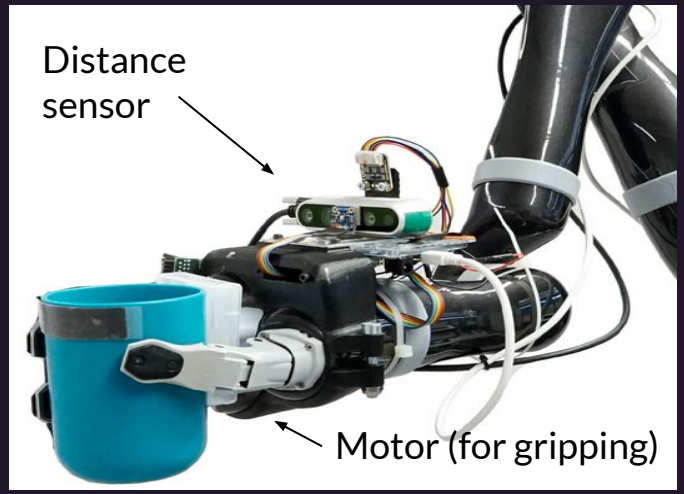


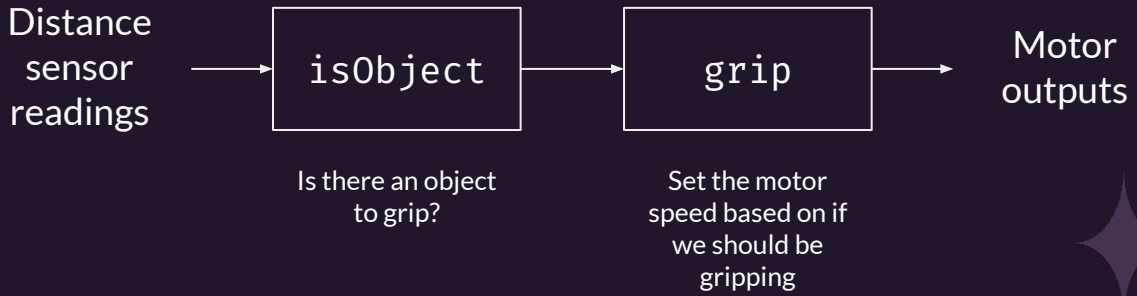
Reactive Programs

A reactive program is one which takes in inputs at each time step (a time delta) to produce outputs at the same rate, as to *react* to those inputs.

We can represent those programs as a *dataflow diagram* to show how the inputs are used at each time step to produce the outputs.

In robotics, we need these programs to be *realtime* (react to inputs without discernible delay) and *fault-tolerant*.

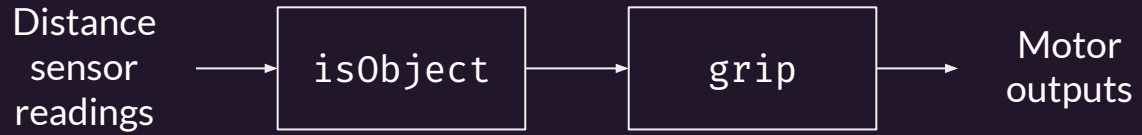




Arrowized Functional Reactive Programming (AFRP)

(as seen in Haskell)

AFRP lets us write code to build these diagrams!




```
arr isObject >>> arr grip
```

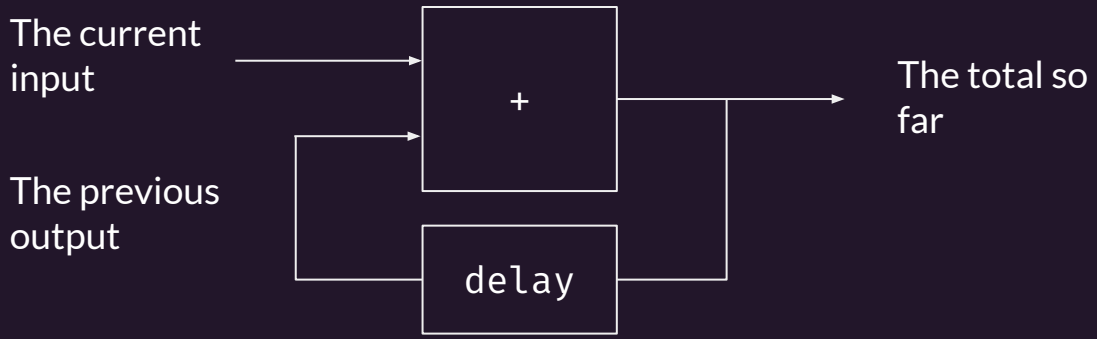
`arr f` build a reactive program by applying `f` to each input it gets to produce each output.

`>>>` composes two reactive programs.

We can now write simple reactive programs where the output at a given time step solely depends on the input at that time step.

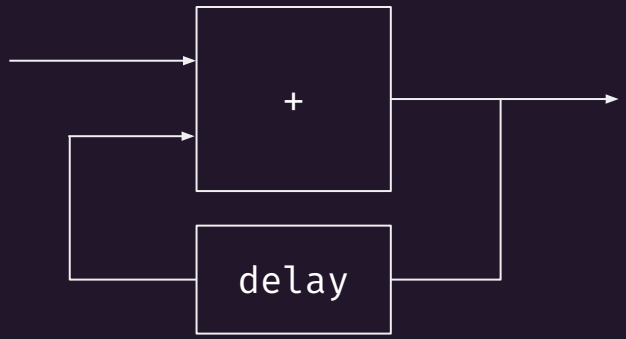
What about cyclic programs?

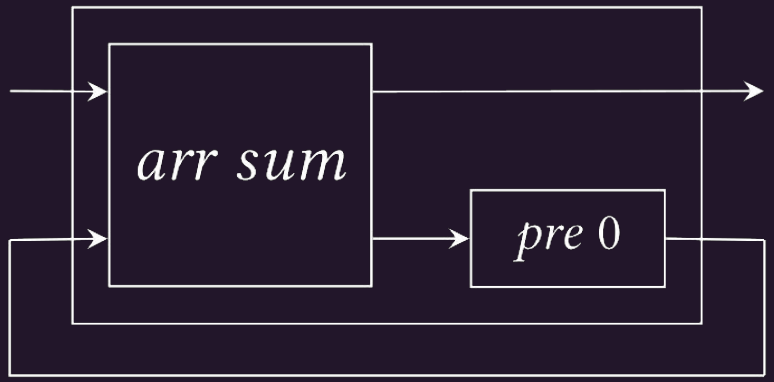
What if we wanted to sum the values of an input over time? You might have a program like this:



How do we implement this in AFRP?

What about cyclic programs?






```
loop (arr sum >>> (id *** pre 0))
```

Creates a loop in the graph

The delay operator - **pre**pends 0 to its input

Don’t worry about the code though

For the rest of the talk, we are just going to look at the diagrams. Any transformation we apply to the diagrams can be applied to our code.

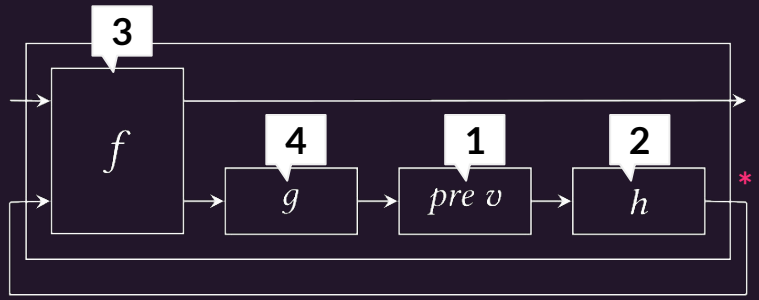
So the takeaway about AFRP is that:

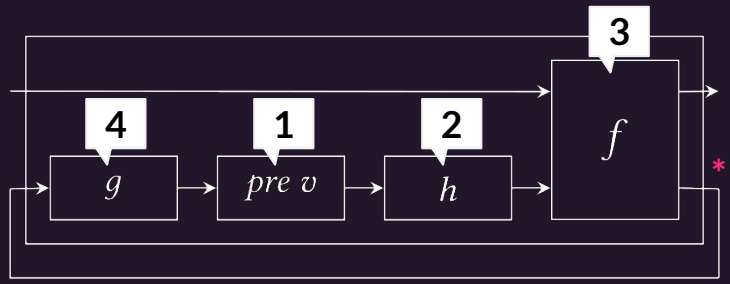
- **You can make basic reactive programs** (`arr` runs a pure function and `pre` is the delay operator)
- **You can compose them in series and/or in parallel** with `>>>` and `***` respectively
- **You can create cyclic diagrams** using `loop`

Why does `loop` drive me loopy?

The execution order of every `loop` differs

It’s not always left to right, and doesn’t always start from the same place…





The plan is to: generate as much of the second output without using the second input as you can, and then use that as the second input. How?

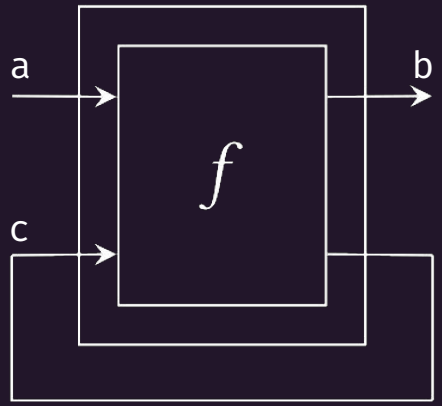
Lazy evaluation lets us evaluate `loop`...

We can define the execution of `loop` in code as:

```
eval (loop f) a =
  let (b, c) = eval f (a, c) in b
```

But the input **c** relies on the output!

Haskell evaluates this using lazy evaluation, where c acts as a promise that it will eventually contain the right value.



...but with performance issues

- Since it is working with a value that it does not know yet, it has to allocate an anonymous function which says “once I get this value, I will be able to produce a result” at every time step.
- This lazy semantics prevents other compile-time optimisations from being applied!

Okay, so what's the aim?

Let's not use lazy evaluation.

We want to transform all `loop` which are possible to execute into a form with known execution order.

How can we make `loop` drive me less loopy?

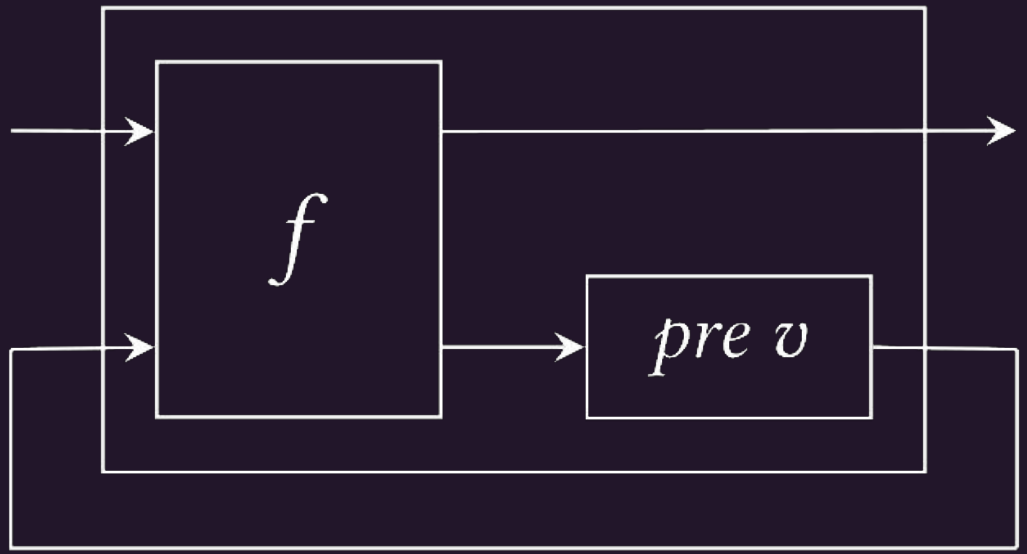
(a whirlwind tour of the tricks we use)

Our target loop form

We aim to transform all loops to the form on the right, called **`LoopD f v`**. (This allows any number of looped values.)

This has trivial execution order so can be run strictly - run `pre v` to get the second output without any input, then run `f`.

We therefore need a set of rules to move `pre` around the `loop` to get to the right position.

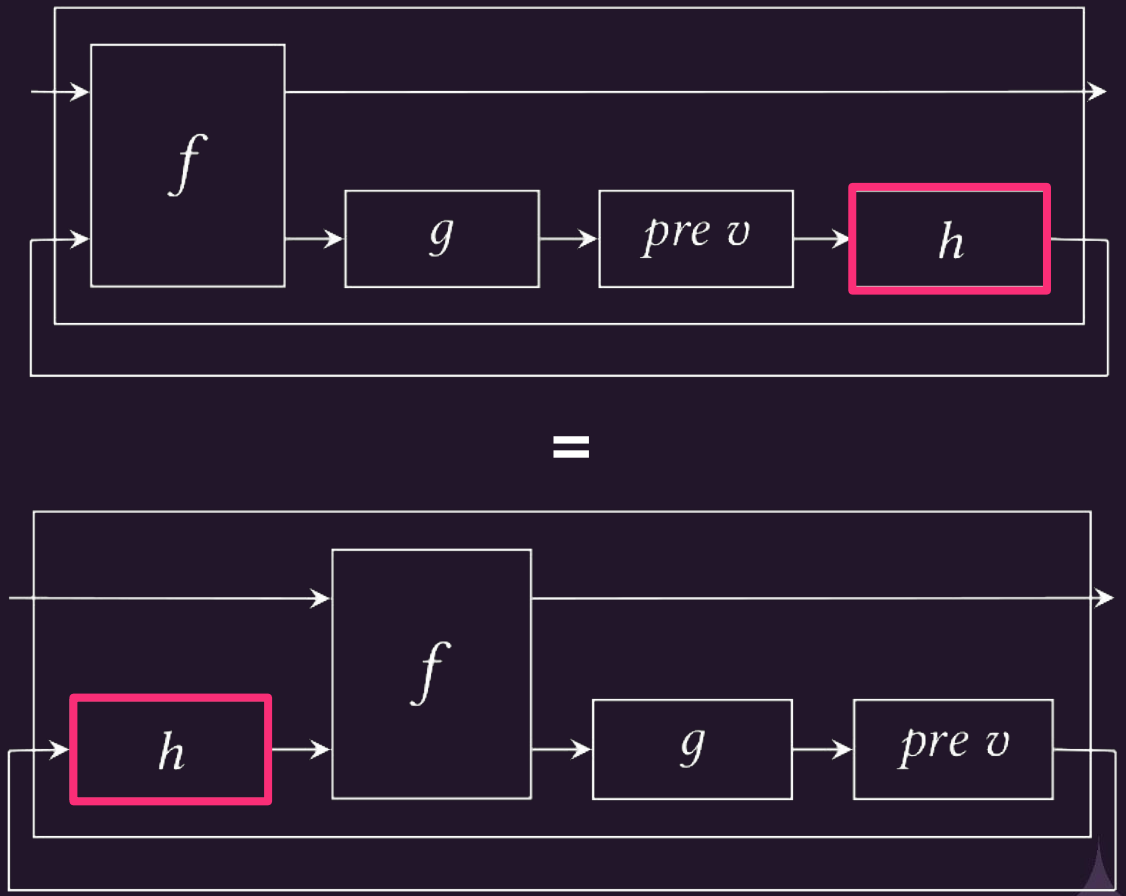


Sliding

This rule takes the entire term in front of the second output and moves it to the back of the `loop`, and vice-versa.

These two diagrams are equal since all of the connections are the same.

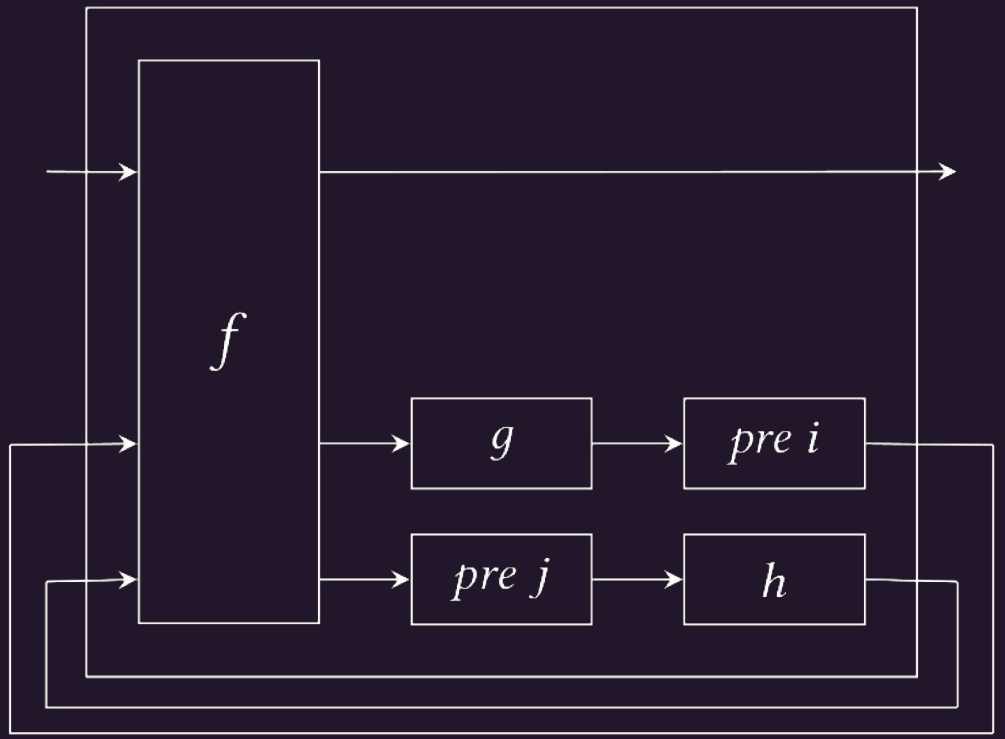
The bottom diagram is a `LoopD`!



Multiple Looped Values

Problem: it's possible to construct graphs where there are multiple values being looped, **each with their own `pre`.**

How can we make this into a `LoopD`? We need there to be a single `pre` somehow.

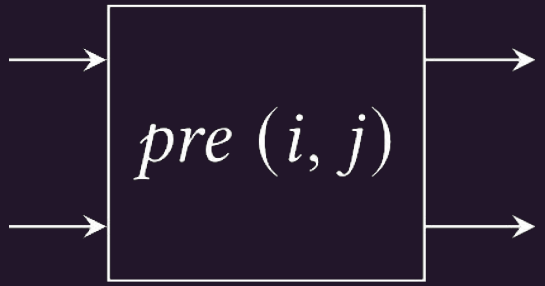


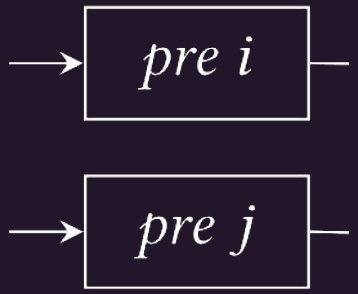

```
loop (f >>> (id *** ((g *** pre j) >>> (pre i *** h))))
```

A nice trick - *product rule*

The *product rule* lets us merge multiple `pre` into a single one - these programs are equivalent.

Our challenge then becomes finding these `pre`, merging them together and then sliding them in the right place.



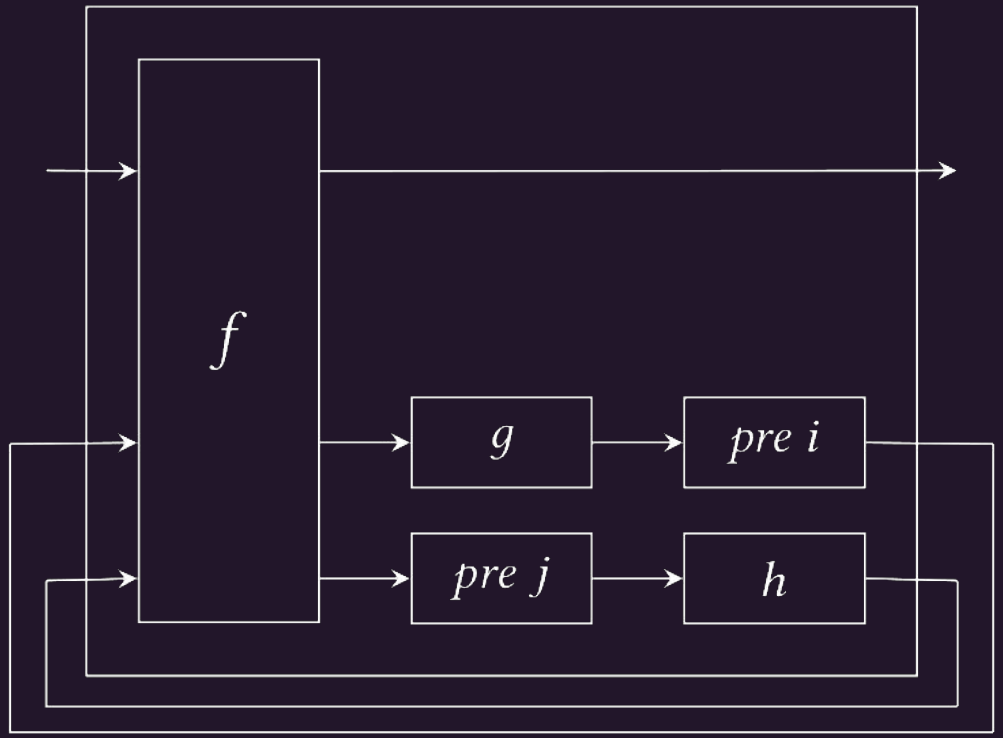



```
pre (i,j) = pre i *** pre j
```

Combining `pres`

We need to merge the two highlighted `pre`, but they currently aren't directly in parallel!

We therefore need to move them so they are in parallel.




```
loop (f >>> (id *** ((g *** pre j) >>> (pre i *** h))))
```

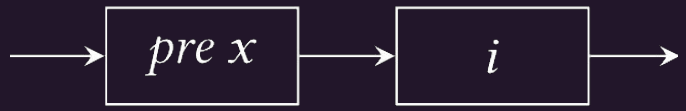
Split

We introduce a new rule, *split*, which does exactly this.

Split attempts to transform an input program into `f >>> d >>> g`, where `d` solely consists of `pre` and `***`.

This consists of three rules.

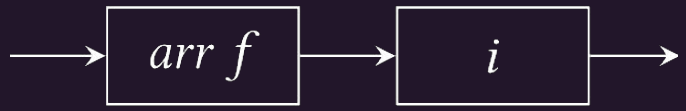
```
split (pre x >>> i) = id >>> pre x >>> i
```



If we find a `pre`, we're done

```
split (arr f >>> i) =
    let x >>> d >>> g = split i
    in (arr f >>> x) >>> d >>> g
```

If we don't, keep going

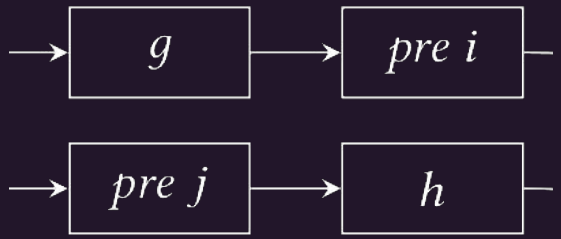



```
split (f *** g) =
    let ff >>> fd >>> fg = split f
        gf >>> gd >>> gg = split g
    in (ff *** gf) >>> (fd *** gd) >>>
(fg *** gg)
```

If we have two parallel paths, solve them individually and align the `pre`

Let’s see this at work!

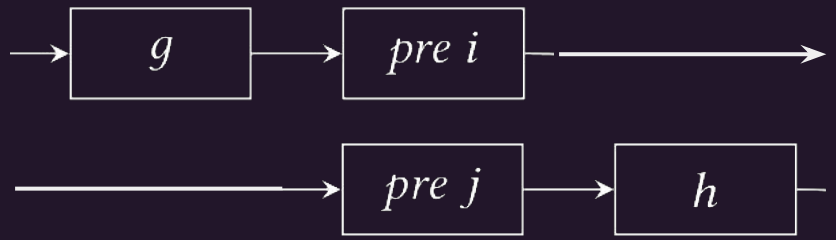
Applying split to part of our program aligns the `pre`!



Let’s see this at work!

Applying split to part of our program aligns the `pre`!

This lets us apply product rule from earlier, then sliding, and then we are done.



sorry for the bad image editing

Let’s put it all together

We apply each of the rules we have seen in an attempt to transform the `loop` into a `LoopD`.

There is an algorithm for this, but in the interests of time it suffices to think of this as a brute-force application of rules until we have a `LoopD`.

Did this actually improve performance?

Yes! Kind of.

We look at the performance of three implementations:

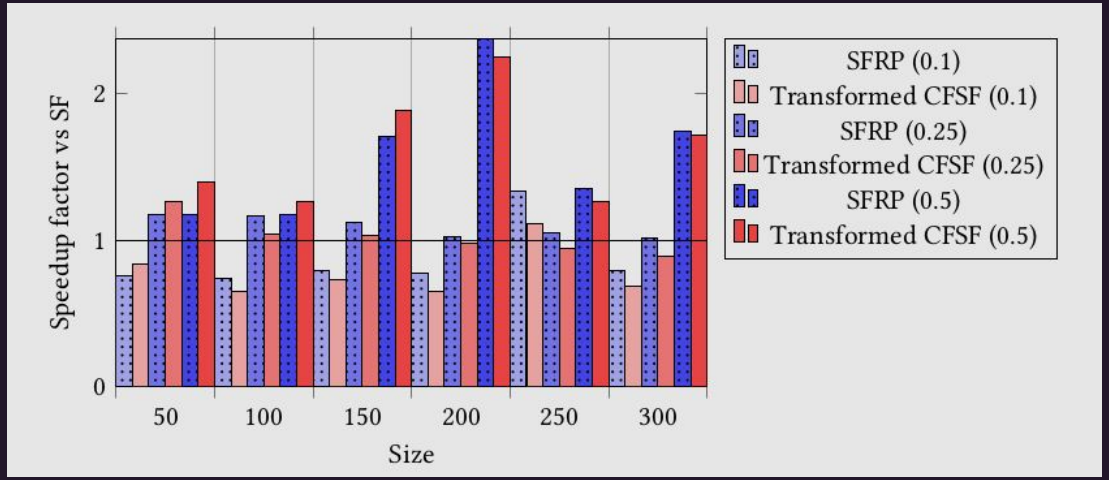
- *Transformed CFSF*: A non-lazy implementation which uses `LoopD` in place of `loop`
- *SFRP*: An implementation which compiles our `LoopD` down to mutable operations on memory
- *SF*: A lazy but heavily optimised implementation (the industry standard)

The results

We look at randomly generated programs of a given size (number of `arr/pre`) and number of loops as a proportion of that given size.

Our performance improves with increased size of program and increased loop proportion!

But there is more to be done.



Conclusions

- We looked at what AFRP is and focused on its `loop` construct, which has unknown execution order.
- We then presented a program transformation which transforms loop into either `LoopD`, which have known execution order.
- We finally took a brief look at some performance results.
- There's more we've done, but didn't have time to cover:
  - Transformations of nested `loop`
  - Proofs that this transformation preserves program meaning, and transforms every `loop`
  - A counterexample that cannot be transformed into a `LoopD`, so we made a variant called `LoopM`
  - Discussion of further optimisations

Thank you!