

PROC-NOTATION

AND YOU,

aka a look into that notation that Jess described as scary and terrible, in which I will hopefully not prove her right!

Seminar run by Finnbar Keating (they/them),
a final year PhD student at Uni of Warwick.

↳ If you have any questions, feel free to
contact me at f.keating@warwick.ac.uk

I hope more of you who were able to attend it
enjoyed the seminar!

A REMINDER OF ARROWS

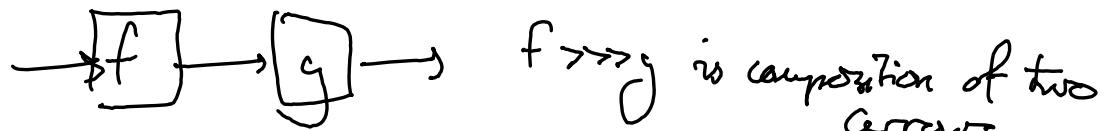
Arrows are composable computations — they have an input and output.

We only need a few constructors to define arrows:



$\text{arr} :: (b \rightarrow c) \rightarrow a \rightarrow c$

arr lets you apply a pure function to values within your arrow!



$f \ggg g$ is composition of two arrows.



first f takes a triple input
and runs f on the first input.

A question that came up in class: what are these actually used for? Here are a few examples:

- Arrowised Functional Reactive Programming (AFRP) which is my expertise: you are building an arrow which takes a stream of inputs to a stream of outputs, thus "reaching" to your inputs. In this domain, arr f maps f across your stream.
- Bidirectional Arrows (Jens knows about this): you are building a program that can be run forwards and backwards. Users don't get access to arr in their domain, but may get e.g. bicarr :: $(a \rightarrow b) \times (b \rightarrow a) \rightarrow a \times b$
- You can also use them to represent data science pipelines.

Arrows are typed as $\text{Arrow arr} \Rightarrow \text{arr } a \rightarrow b$, where a is the type of inputs and b the type of outputs.

e.g. $\text{arr (+1)} :: (\text{Arrow arr}, \text{Num } a) \Rightarrow \text{arr } a \rightarrow a$

QUESTION How can you define second (like first but on the second element)?

[$\text{second } f = \text{arr swap} \ggg \text{first } f \ggg \text{arr swap}$
where $\text{swap } (x, y) = (y, x)$.]

Note how we used arr to route data!]

PROC NOTATION

Let's look at a silly program:
This program is very silly, but
it'll hopefully prove a point.

silly(x,y,z):
$w = f(x,z)$
$p = g(y,w)$
return (x,p)

Okay let's try to write this.

(x,y,z) we need
(x,z) and (y,w)
arr ($\lambda(x,y,z) \rightarrow ((x,z),(y,w))$)

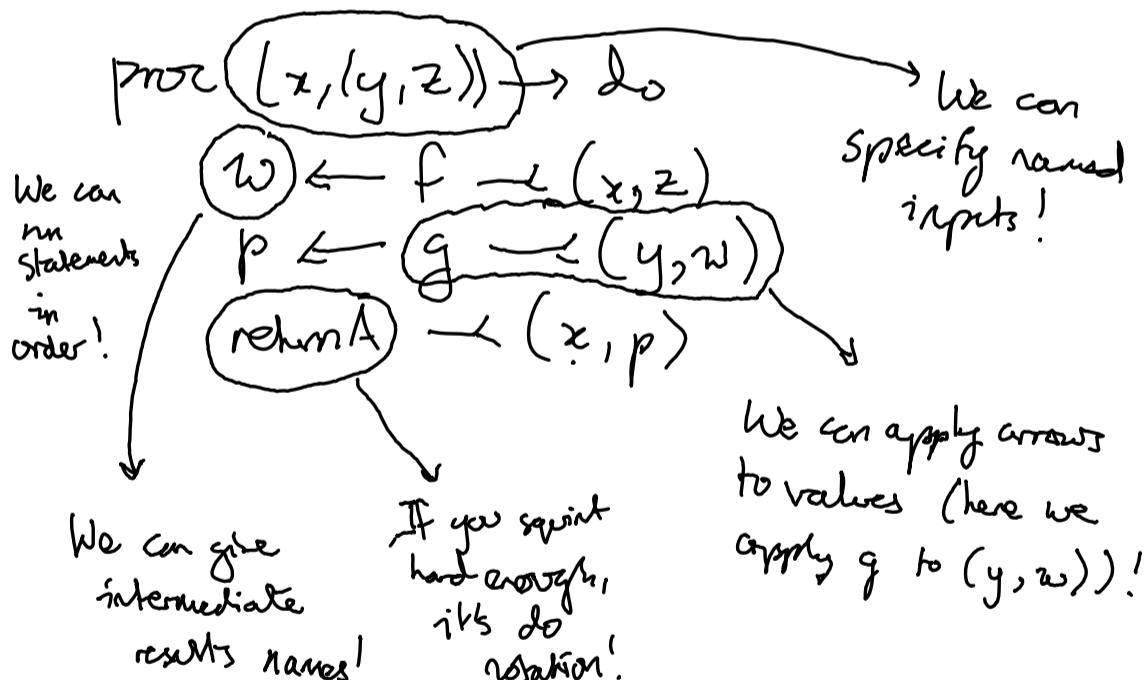
>>> first f. so now we have (w, (y,x))

>>> arr ($\lambda(w,y,x) \rightarrow ((y,w),x)$)

>>> first g so now we have (p,x)

>>> arr sup

I deliberately didn't finish this example in the seminar,
because it's awful*. Luckily, there is an alternative to
writing that out manually — proc notation:



And! It'll automatically translate to uses of the arrow combinators (the horrible thing I was starting to do).

* Fun fact: when I wrote this example out in my notes, I actually made a mistake with the denotation that I didn't notice until the day of the seminar. I think this just further proves the benefits of proc.

TODAY : THAT TRANSLATION

We're going to look at how that proc-notation code is translated to the arrow combinators,
in a series of puzzles in slowly increasing difficulty!
(It's me, a Zetetics level designer (I wish))

For each puzzle, provide the equivalent code using only arrow combinators. e.g. for PUZZLE 2, your code should take in an input of form (x,y) and produce the output consisting of running $f \dashv y$. Answers at the end.

PUZZLE 1
(5 picarats*)

Proc $x \rightarrow \text{returnA} \dashv x$

PUZZLE 2
(10 picarats)

Proc $(x,y) \rightarrow f \dashv y$

QUESTION 1
How do we compile
Proc $a \rightarrow f \dashv c$
in general?

[Intuition: the general idea is that we want to take the variables we have and filter them down to more in c using arr , and then pass them to f .]

* If you're confused what a Picarat is, it's the currency that you get for solving puzzles in Professor Layton. Also Jen thinks that no puzzles offer only 5 Picarats and that the minimum is 10. We couldn't find out who was right.

NOW, MULTIPLE STATEMENTS

PUZZLE 3 $\text{proc } x \rightarrow \text{do}$

(10 points) $y \leftarrow f \rightarrow x$

$\text{returnA} \rightarrow (x, y)$

[Hint: you need a way to route x to both the first and second line.]

PUZZLE 4 $\text{proc } x \rightarrow \text{do}$

(15 points) $y \leftarrow f \rightarrow x$

$z \leftarrow g \rightarrow x$

$\text{returnA} \rightarrow (y, z)$

QUESTION 2 How do we compile

$\text{proc } x \rightarrow \text{do}$

$y \leftarrow f \rightarrow e$

$\text{returnA} \rightarrow (y, e')$

for some e, e' which are defined in terms of x ?

[That is, $\lambda x \rightarrow e$ is a well-defined function, and similar for e' — pretend I'm going to replace e and e' with e.g. $x + 3$.]

QUESTION 3 How do we compile

(harder, we didn't
cover this.)

$\text{proc } x \rightarrow \text{do}$

$y \leftarrow f \rightarrow e$

CS

Assuming we can compile CS into
 CS^* with the input (y, e') ?

[Equivalently, the answer will be of the form

(some code that generates (y, e')) $\ggg CS^*$]

Phew, that's most of the stuff you'll be doing with proc!

AN ASIDE:

HOW DOES HASKELL DO THIS?

Question that came up in the seminar: how does Haskell approach the denugaring of do with proc in general?

Answer: at each line, the variables being carried around are split into those needed by the current line and those needed in future. e.g.

proc (x,y) → do
* z ← f ← x
return ← (z,x,y)

At part *, we generate

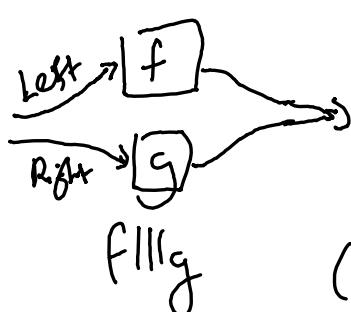
arr ($\lambda(x,y)$) → $(x, (\lambda(x,y)))$)
 ↑
 needed
 by this
 line
 ↑
 needed on
 future lines

Lots of the definitions we came up with used arr drop, which also works!

HEY, WHY NOT \dagger CONDITIONALS

We didn't cover this stuff, but I've left it in because you might be interested!

Let's introduce ArrowChoice:



$(\parallel\parallel) :: \text{ArrowChoice} \ a \Rightarrow$
 $a \ b \ c \rightarrow a \ b' \ c \rightarrow$
 $a \ (\text{either } b \ b') \ c$

(NB: ArrowChoice is defined in terms of other operators, but we focus on $\parallel\parallel$ because it makes the definition simpler.)

We also get a new form within our pure notation:

proc $x \rightarrow$ if even x then $f \multimap x$
 else $g \multimap x$

A value to check
 (this isn't an arrow!) Conditional arrows to run.

PUZZLE 5 The above definition.

(10 picarats)

[Hint: you will need an arrow which produces an Either value.]

PUZZLE 6

(15 picarats)

proc $(x,y,z) \rightarrow$ if x then $f \multimap y$
 else $g \multimap z$

QUESTION

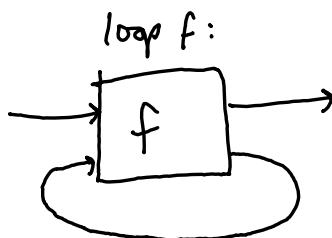
4

Assume that c , e_t and e_f are all written in terms of x , so if we have x in scope you can write c , e_t and e_f .

How do we translate

proc $x \rightarrow$ if c then $t \multimap e_t$
 else $f \multimap e_f$?

MY NEMESIS, LOOP



loop :: ArrowLoop a \Rightarrow
a (b,c) (c,d) \rightarrow a b c

This takes an arrow that runs on two inputs to produce two outputs, and sets it so that the second output is used as the second input.

OPTIONAL ASIDE
This probably seems a bit odd — in what contexts would this not infinite loop? If you're curious, you can check out the introduction to the paper

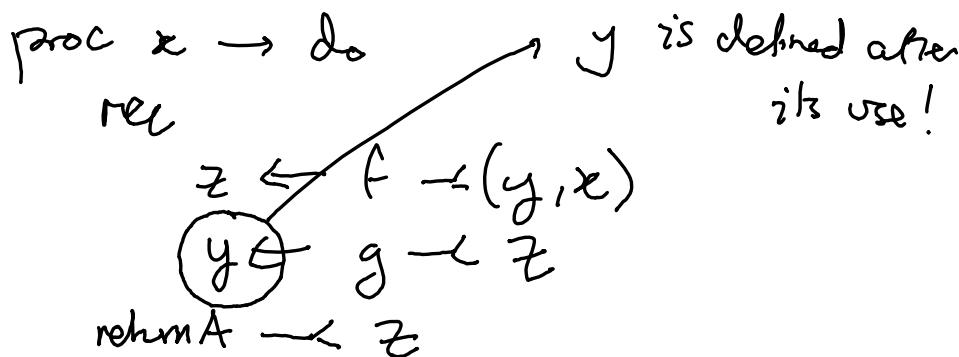
| This Is Driving Me Loopy: Efficient Loops in
| Arrowized Functional Reactive Programs

[Keating and Hale, published in Haskell '13]

which should hopefully be an accessible look at loop in the domain of Arrowized Functional Reactive Programming.

There is also a video, if you prefer to learn that way and aren't fed up of my presenting style yet.

loop gives us access to recursive definitions — variables defined after their use:

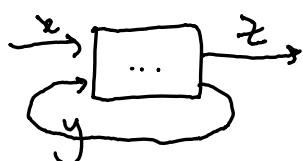


QUESTION Why did I show you loop when introducing rec? How do they relate?

(The answer to this is on the next page — pause for a moment to think about it and then move on when you've satisfied.)

ANSWER

loop lets us pass a value back to the start. We're



effectively writing a loop of the form on the left — the proc has an input of x and output of z , and since y is needed before it is defined, we need to pass it back to the start of the arrow being looped.

PUZZLE 8 proc $x \rightarrow$ do

(20 Picarats,
my sanity)

rec

$$z \leftarrow f \dashv (y, x)$$

$$y \leftarrow g \dashv z$$

returnA $\rightarrow z$

This will take a while.

[Hint: your program will be of the form

loop (arr ($\lambda(x, y) \rightarrow \dots$) $\ggg \dots \ggg$ arr($\dots \rightarrow (z, y)$))]

QUICK CONCLUSIONS

Arrows are a useful abstraction in a variety of contexts, but writing large programs only using the arrow combinators (`arr`, `○` and `first`) is time-consuming and error-prone (to put it politely).

So we have `proe` notation, which does the various horrible bits for moving data for you in the background. Nice!

We stepped through what that looks like, and how it is transformed into code which uses the arrow combinators — while you will hopefully never have to perform this translation yourself, it's useful to see how it is done!

Anyway, I hope you found this interesting — thank you for reading!

ANSWERS

P1: it's just id

P2: arr snd >>> f

Q1: arr ($\lambda a \rightarrow e$) >>> f

P3: arr dup >>> first f >>> arr swap

where dup $x = (x, x)$

N.B. a student came up with the better answer
arr dup >>> second f

Marshall won't spot this!

P4: arr dup >>> first f >>> arr ($\lambda(y, x) \rightarrow (x, y)$)

>>> first g >>> arr ($\lambda(z, y) \rightarrow (y, z)$)

Or the student-provided:

arr dup >>> first f >>> second g

Therefore proving that students are better than ATC (?)

Q2: arr ($\lambda x \rightarrow (e, e')$) >>> first f

This was correctly expressed by a student as:

arr dup >>> first (arr zs >>> f) >>>

second zs

where $e = \lambda x$ and $e' = \lambda x$

- so rather than doing the handwaving that I did with e and e' , they defined them as functions of x called zs and zs . Nice!

Q3: arr ($\lambda x \rightarrow (e, e')$) >>> first f >>> cst

P5: arr ($\lambda x \rightarrow \text{if even } x \text{ then Left } x \text{ else Right } x$)
>>> (f ||| g)

So, produce a Left/Right based on the condition and then do f ||| g.

P6: arr ($\lambda(x, (y, z)) \rightarrow \text{if } x \text{ then Left } y \text{ else Right } z$)
>>> (f ||| g)

Here the Left/Right differ slightly due to the arguments needed by f and g.

Q4: arr ($\lambda x \rightarrow \text{if } c \text{ then Left } e_L \text{ else Right } e_R$)
>>> (t ||| f)

P7: loop (arr ($\lambda(x, y) \rightarrow ((y, x), ())$)
>>> first f >>> arr ($\lambda(z, ()) \rightarrow (z, z)$)
>>> first g >>> arr ($\lambda(y, z) \rightarrow (z, y)$))