APL: Sheet 5

Hopefully you know the drill by now

① i)

$$\dfrac{\dfrac{\dfrac{}{n : Nat \vdash n : Nat}\,VAR}{n : Nat \vdash succ(n) : Nat}\,SUCC}{\vdash fix(n : Nat . succ(n)) : Nat}\,FIX$$

ii) $\Gamma_1 = f : Nat \to Nat \to Nat$
$\Gamma_2 = \Gamma_1 \cup \{n : Nat\}$
$\Gamma_3 = \Gamma_2 \cup \{m : Nat\}$
$\tau = Nat$
$\Gamma_4 = \Gamma_3 \cup \{x : Nat\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_4 \vdash f : Nat \to Nat \to Nat}\,VAR \quad \dfrac{}{\Gamma_4 \vdash n : Nat}\,VAR}{\Gamma_4 \vdash f(n) : Nat \to Nat}\,APP \quad \dfrac{}{\Gamma_4 \vdash x : Nat}\,VAR}{\Gamma_3, x : Nat \vdash f(n)(x) : Nat}\,APP}{\Gamma_3, x : Nat \vdash succ(f(n)(x)) : Nat}\,SUCC$$

application associates to the left

$$\dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash m : Nat}\,VAR \quad \dfrac{}{\Gamma_3 \vdash n : \tau}\,VAR \quad \Gamma_3, x : Nat \vdash succ(f(n)(x)) : Nat}{\Gamma_3 \vdash ifz(m; n; x.succ(f(n)(x))) : Nat}\,IFZ}{\Gamma_2 \vdash \lambda m : Nat . ifz(m; n; x.succ(f(n)(x))) : Nat \to Nat}\,LAM$$

$$\dfrac{\Gamma_1 \vdash \lambda n : Nat . \lambda m : Nat . ifz(m; n; x.succ(f(n)(x))) : Nat \to Nat \to Nat}{fix\,(f : Nat \to Nat \to Nat . \lambda n : Nat . \lambda m : Nat . ifz(m; n; x.succ(f(n)(x))) : Nat \to Nat \to Nat}\,LAM\,FIX$$

$$\vdash plus : Nat \to Nat \to Nat \quad \text{def. plus}$$

② Following the same pattern as the previous sheets, we will next get to know the dynamics by exploring transition sequences

(i) I'm going to be mean and say this is just boilerplate - boring and annoying boilerplate - so I won't write it out.

If you struggle due to the precision required in copying from line to line, don't worry, you're a human of course this is hard.

If you are struggling with the concept of these transitions go back to worksheet 3. Like I said above we are just doing the same process each week, just with more complex rules as each week a new concept is added to our little language.

(ii) This part is interesting because having added recursion, we can get things that infinitely loop:

$$
\begin{aligned}
& \lambda x(x:\mathrm{Nat}.x) \\
\mapsto & \quad \{\mathrm{D\text{-}Fix}\} \\
& \mathrm{letrec}\,[\lambda x(x:\mathrm{Nat}).x\,/\,x] \\
\equiv & \quad \{\mathrm{S\text{-}Lbst}\} \quad \text{(watch this is a } \equiv \text{ not } \mapsto\text{)} \\
& \lambda x(x:\mathrm{Nat}.x) \quad \text{and we are back where we started} \Rightarrow \text{this will loop:}
\end{aligned}
$$

$$
\begin{aligned}
\mapsto & \quad \{\mathrm{D\text{-}Fix}\} \\
& \mathrm{letrec}\,[\lambda x(x:\mathrm{Nat}).x\,/\,x] \\
\equiv & \quad \{\mathrm{S\text{-}Lbst}\} \\
& \lambda x(x:\mathrm{Nat}.x)
\end{aligned}
$$

$$
\begin{aligned}
\mapsto & \quad \{\mathrm{D\text{-}Fix}\} \\
& \mathrm{letrec}\,[\lambda x(x:\mathrm{Nat}).x\,/\,x] \\
\equiv & \quad \{\mathrm{S\text{-}Lbst}\} \\
& \lambda x(x:\mathrm{Nat}.x)
\end{aligned}
$$

...

③ Now we are really looking at the point that you must be careful to have things resolve. Last Q we had a looping term, now we see a term that grows.

(i) $fix\,(x{:}Nat \to succ(x))$

$\mapsto$ {D-FIX}

$succ(x)[fix\,(x{:}Nat \to succ(x))/x]$

$\equiv$ {subst}

$succ(fix\,(x{:}Nat \to succ(x)))$

$\mapsto$ {D-SUCC, D-FIX}

$succ(succ(x))[fix\,(x{:}Nat \to succ(x))/x]$

...

We will just keep infinitely succing because D-SUCC is the only rule we can match on from this point.

(ii) This fixes the aforementioned problem as we can pick VAL-SUC instead:

$fix\,(x{:}Nat \to succ(x))$

$\mapsto$ {D-FIX}

$succ(x)[fix\,(x{:}Nat \to succ(x))/x]$

$\equiv$ {subst}

$succ(fix\,(x{:}Nat \to succ(x)))$

value by VAL-SUC.

iii) I wouldn't worry if you didn't get this Q, I think it is more to just expand your horizons/ make you think.

Now, if you know Haskell, you will know that normalises to

"Weak head normal form"

meaning that the outermost part gets evaluated to a constructor of a lambda. In other words we always know the shape of a value and after that we don't care.

⇒ in this example, Haskell will always encrep the first succ making it most similar to revised rule

$$\frac{}{\text{succ}(e)\ \underline{\text{val}}}\ \text{VAL-SUCC}$$

(

encrep outermost constructor, then doesn't care (indicated by a wild card).

However, that is just Haskell's internal action, justifying why it is happy with terms like

```
> infnat = succ infnat
```

what happens next depends on the evaluation you want to perform on your data type.

We can make Haskell behave like the original rule by demanding to encrep the whole value:

```
>instance Show Nat where
>    show Zero     = "Zero"
>    show (Succ n) = "Succ (" ++ show n ++ ")"
```

forces n

Or we can maintain the current behaviour by explicitly not caring about more than the information constructor:

```
>instance Show Nat where
>    show Zero     = "Zero"
>    show (Succ _) = "Succ <who cares>"
```

(4) Sweet, first part of this is just programming in Haskell!

The hint of the Q also pushes us towards the FP trope that multiplication is just repeated addition

2 × 3 = 2 + 2 + 2 = 6

This is a classic PL question as we are defining new ops (e.g. times) in terms of old ops as it allows for a small core language. So if stuck on such a Q with no hint and no previous knowledge think: how can I define this using what I already have.

Anyways, writing a Haskell function follows this - hopefully familiar - process:

- write name
- write type
- pattern match
- always let the types be your guide :)

(I've uploaded my Haskell function recipe sheet that I give to the first years if you are interested)

```
> times :: Nat -> Nat -> Nat
> times n zero        = zero               -- 0 x n = 0
> times n (Succ zero) = n                  -- 1 x n = n
> times n (Succ n)    = plus (times n x) n
                        -- multiplication is repeated addition.
```

But we are not here to teach you Haskell, we hope that by now we have done that and the above is just there as a reminder for completion. The reason we ask for Haskell is to encourage you to use a familiar touch

stare when writing PCF functions, because we can convert Haskell to PCF using the following steps:

- recursion => we need fix
- pattern matching on nats => we need ifz

times $\stackrel{def}{\equiv}$

fix (f : Nat -> Nat -> Nat. -- recursive point (same ty as times)
  λn.
  λm.] two args
  -- now we use our ifz combinator to pattern match
    ifz(m; -- who we pattern matching on
z case: zero;
S case: x.
(predecessor who we wanna also pattern match on:)
      ifz (z; -- subject of match
z case: n;
S case: y. plus (f(n)(x)) (n)))) : Nat -> Nat -> Nat

y: pred (who we can name as we want x)

f(n)(x): recurse

plus (...) (n): same action as Haskell

))) : Nat -> Nat -> Nat: close all brackets + give ty

(5) New lang with a new concept, which means we need to explore our classic properties of:
- subst
- inversion
- progress
- preservation

(i) Read the Q carefully, it wants us to state and prove.

Recall general subst lemma,

In prose it says:

"Say we have a term that uses a variable x in it. If I swap out the variable x for an expression of the same type, the type of my overall expression shouldn't change"

If $\Gamma \vdash e : \tau$ and $\Gamma, x:\tau \vdash u : \sigma$
then $\Gamma \vdash u[e/x] : \sigma$

swap in our e of the same ty as x — won't change ty of u.

Since this says nothing specifically relating to FCF and PCF uses the same notation for statics, this statement of subst remains unchanged

All that remains is to prove it, and the Q is kind enough to only make us do so for the fix case, representing the new concept of the week.

As before, this is a

Proof by induction on $\Gamma, x:\tau \vdash u : \sigma$

(most interesting of our two premises)

[Case · Fix] for interest I have recouped about this in a different circle than usual

Let's remind ourself of this rule, and unpack what it gives us:

$$\frac{\Gamma, x:\tau \vdash e:\tau}{\Gamma \vdash fix(x:\tau.e):\tau}\ Fix$$

normally I blindly unpack and pray it's right. This time I use my dependence to know I will want to use the IH, so I actively seek the obligations needed to use it.

One thing on top of some shape $\Rightarrow$ we will have an IH.

Because what we are inducting on has an $x:\tau$ in its env, I will rephrase our rule so as to conserve less by having no name conflicts:

$$\frac{\Gamma, y:\tau \vdash b:\tau}{\Gamma \vdash fix(y:\tau.b:\tau)}\ Fix$$

I've also renamed e to be b for body to avoid confusion

Now I will write out my IH, which is an exact rephrasing of what we are trying to prove but for the premise of the rule:

What we trying to prove:

If $\Gamma \vdash e:\tau$ and $\Gamma, x:\tau \vdash u:\sigma$
then $\Gamma \vdash u[e/x]:\sigma$

Specialized to premise (making it IH):

Specializations:

$u = b$

$\Gamma = \Gamma, y:\tau$

IH = If $\Gamma \vdash e:\tau$ and $\Gamma, y:\tau, x:\tau \vdash b:\sigma$
then $\Gamma, y:\tau \vdash b[e/x]:\sigma$

In order to use this IH, we will need to discharge its obligations.

Luckily it's first is the same as the antecedent we are trying to prove, so we can assume it

$$ASS = \Gamma \vdash e : \tau$$

The others we can deduce from our case:

In order to have an expression using fix, we must have used the fix rule giving us the following derivation tree:

$$\dfrac{\vdots \text{SHAPE} \qquad \Gamma, y:\tau \vdash b : \tau}{\Gamma \vdash fix(y:\tau.b:\tau)}\ Fix$$

which is almost our second obligation bar one extra variable, which we can get through weakening.

Thus I can construct a derivation of my goal as follows:

$$\dfrac{\dfrac{ASS}{\Gamma \vdash e:\tau} \qquad \dfrac{\dfrac{\vdots \text{SHAPE}}{\Gamma, y:\tau \vdash b:\sigma}}{\Gamma, y:\tau, x:\tau \vdash b:\sigma}\ WK}{\dfrac{\Gamma, y:\tau \vdash b[e/x]:\sigma}{\dfrac{\Gamma \vdash fix(y:\tau.b[e/x]):\sigma}{\Gamma \vdash fix(y:\tau.b)[e/x]:\sigma}\ \begin{array}{l}def.\ subst.\\ (assuming\ y \neq x)\end{array}}\ Fix}\ IH$$

(Free very helpful to ensure I discharge all proof obligations)

□

(ii) Just state.

Here we just look at the Fix rule and assert that it is the only way to have made an expression featuring fix

$$\frac{\Gamma, x:\tau \vdash e:\tau}{\Gamma \vdash fix(x:\tau.e):\tau}\ Fix$$

Inversion for Fix:

Suppose $\Gamma \vdash e:\tau$.

If $e = fix(x:\tau'.e)$

then it must be that

$\tau = \tau'$ (fix puts itself inside itself, so its and its same as the whole term)

$\Gamma, x:\tau \vdash e:\tau$ (just premise)

(iii) State and prove progress and preservation

When it comes to stating both statements remain the same as they rely on type judgements in general and nothing specific about the expression:

Statements:

Claim One (Preservation):

If $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$

"evaluation doesn't change the type"
"the type is preserved"

Claim Two (Progress):

If $\vdash e : \tau$ then either eval or $e \mapsto e'$ for some e'

"when a term is typable it is either a value or something we can do work on"
"Progress can always be made on a typable term until it is a value."

Proofs:

Hopefully we know the skills on proving these statements by now. Remember we only need to prove for the fix case.

Proof of Preservation:

Proof by induction on $e \mapsto e'$.

(as inducting on that premise gives us access to e')

[Case: D-Fix]

$\Rightarrow e \to e'$ has shape and derivation:

$$\frac{}{fix(x:\tau.e) \mapsto e[fix(x:\tau.e)/x]}\ \text{D-FIX}$$

This rule has no premise so is hopefully easy to prove. That, or we need to remember something from elsewhere.

$$\begin{aligned} \text{GOAL} &= \vdash e' : \tau \\ &= \vdash e[fix(x:\tau.e)/x] : \tau \end{aligned}$$

Reminding myself of the goal, I see it's the latter. This is a matter of substitution.

By the substitution lemma:

If $\Gamma \vdash e : \tau$ and $\Gamma, x:\tau \vdash u : \sigma$
then $\Gamma \vdash u[e/x] : \sigma$

Oh and we need inversion to discharge the antecedents:

And inversion on Fix:

If $e = fix(x:\tau'.e)$
then it must be that $\tau = \tau'$ and $\Gamma, x:\tau \vdash e : \tau$

The shape of e is $fix(x:\tau'.e)$, meaning that by inversion we have $x:\tau$ and $\Gamma, x:\tau \vdash e : \tau$, allowing us to apply the substitution lemma to obtain our goal of $\vdash e[fix(x:\tau.e)/x] : \tau$. □

Proof of progress:

Proof by induction on $\vdash e : \tau$

(only thing we can induct on as there is only one premise.)

[Case: FIX]

⇒) $te:\tau$ has shape and derivation: $\vdash fix(x:\tau.e):\tau$

$$\frac{x:\tau \vdash e:\tau}{\vdash fix(x:\tau.e):\tau}\ FIX$$

IH = If $x:\tau \vdash e:\tau$ then either eval or $e \mapsto e'$ for some e'

GOAL = $fix(x:\tau.e):\tau$ val

OR

$\exists e'$ s.t. $fix(x:\tau.e):\tau \mapsto e'$

From our experience, this will go one of two ways: either the rules will be bad and we can conclude immediately one of the goals using premise, or we will need to case split on the IH.

Let's see if any of the fix rules have a premise...

Sadly there is no rule that immediately makes fix a value, BUT there is a rule that always reduces a fix: D-FIX.

By D-FIX $\exists e'$ s.t. $e \mapsto e' : e[fix(x:\tau.e)/x]$

□

(6) (i) Honestly, I don't remember what uniqueness of typing is, so my first action is to literally the sign posting of the Q and look back at week 2.

Type uniqueness:

if $\Gamma \vdash e : \sigma$ and $\Gamma \vdash e : \tau$
then $\sigma = \tau$

"one type per program"
"types are unique"

lol that's it for (i): just look back and copy claim 1.

(ii) Since both premises are the same bar renaming, it doesn't matter which one we induct on.

Proof by induction on $\Gamma \vdash e : \tau$

[Case: FIX]

$\Rightarrow \Gamma \vdash e ; \sigma$ has shape $\Gamma \vdash fx(x:\tau.e) : \tau$ and derivation:

$$\frac{\Gamma, x:\tau \vdash e : \tau \quad :\text{SHAPE}}{\Gamma \vdash fx(x:\tau.e) : \tau}\ \text{FIX}$$

IH = if $\Gamma, x:\tau \vdash e : \tau$ and $\Gamma, x:\tau \vdash e : \sigma$ then $\sigma = \tau$

GOAL = $\sigma = \tau$

ASS = $\Gamma \vdash fx(x:\tau.e) : \sigma$

From inversion on ASS we have $\tau = \sigma$.

(iii) No. Proof by counter example:

Multiple types can be assigned to $\beta x.(\beta c.c)$:

Basically whatever you say the fn is goes as if goes straight into the end, allowing application of VAR.

$$\dfrac{\dfrac{}{x:Nat \vdash x:Nat}\,VAR}{\vdash \beta x.(\beta c.c):Nat}\,Fx \qquad \dfrac{\dfrac{}{x:Str \vdash x:Str}\,VAR}{\vdash \beta x.(\beta c.c):Str}\,Fx$$