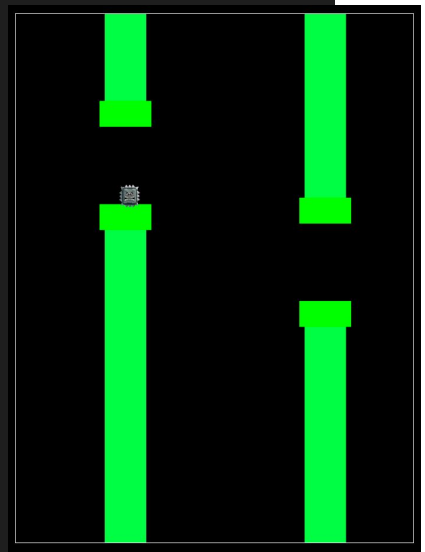
Functional Postfix Lang

Postfix Haskell

- Function overloading only form of branching
- N-dimensions of subtyping
- Almost sum types

Desktop > flappy > src > gate.phs

```
"../node_modules/postfix-haskell/planning/stdlib/prelude.phs" require use
"../node_modules/postfix-haskell/planning/stdlib/static_mem.phs" require $mem =
"./engine.phs" require $game =

## Gate - combination of a top pipe and bottom pipe
(   I32 # x - position on the screen
    F32 # y - height of the opening
) class $Gate =

# Gate constant dimensions
150 $GAP =          # Gap between top and bottom pipes
100 $WIDTH =        # Width of the pipe
10 $LIP_WIDTH =     # Lip overhang
50 $LIP_HEIGHT =    # Height of the lip sections
2 $SPEED =          # How fast the gates move across the screen

((Gate) (Gate): # Move gate to the left
    unpack ( $x $y ) =
    ( x SPEED - y ) Gate make
) $update =

# Randomly generate a gate
((()(Gate):
    # Make a gate with randomized hole height
    (   game.WIDTH
        game.rand game.HEIGHT GAP - F32 cast *
    ) Gate make
) $respawn =

# Overload operators for static memory
((Gate I32): 1 ) (:
    $addr = unpack ( $x $y ) =
    x addr mem.static_init
    y addr 4 + mem.static_init
) $mem.static_init fun
((Unit Gate I32): 1 ) (:
    $addr = unpack ( $x $y ) =
    x addr mem.store
    y addr 4 + mem.store
) $mem.store fun

# Rectangular Hitboxes
((Gate): unpack ( $x $y ) =
    x
    0
    WIDTH
    y
) $hitbox_top =
((Gate): unpack ( $x $y ) =
    x
    y GAP +
    WIDTH
    game.HEIGHT y -
) $hitbox_bot =

((Unit Gate): # Draw gate onto canvas
    unpack ( $x $y ) =
    # Top pipe
    x LIP_WIDTH +
```



```
##
# Operator *
##

# Mul values of same type
((I32 I32): 1 ) (: "i32.mul" asm ) $global.* fun
((I64 I64): 1 ) (: "i64.mul" asm ) $global.* fun
((F32 F32): 1 ) (: "f32.mul" asm ) $global.* fun
((F64 F64): 1 ) (: "f64.mul" asm ) $global.* fun

# Attempt to promote values to same type
$_can_promote ~ (: _promote * ) $global.* fun

# Optimization for identity property of multiplication
(: pop _is_1 ) (: swap pop ) $global.* fun
(: _is_1 ) (: pop ) $global.* fun

# Negation
(: pop _is_neg1 ) (: swap pop neg ) $global.* fun
(: _is_neg1 ) (: pop neg ) $global.* fun

# Optimization for zero property of multiplication
# TODO? maybe make the zero the same type?
(: pop _is_0 ) (: pop pop 0 ) $global.* fun
(: _is_0 ) (: pop pop 0 ) $global.* fun

# TODO optimize to shl
```

$$\tau ::= \text{I32} \mid \text{I64} \mid \text{F32} \mid \text{F64} \mid (\tau^*) \mid \text{Unit} \mid \tau_1\ \tau_2\ \text{Arrow}$$
$$\mid \quad e\ \text{type} \mid e\ \$\text{T} = \mid \text{T} \mid \tau\ \text{unpack}$$

$$e ::= e\ e + \mid e\ e \times \mid e\ e\ e\ \text{select} \mid e\ e\ == \mid \tau\ \tau\ ==$$
$$\mid \quad (e^*) \mid ((\tau^*) : e^*) \mid e\ \tau\ \text{fix} \mid \overline{n}_{\text{I32}} \mid \overline{n}_{\text{I64}} \mid \overline{n}_{\text{F32}} \mid \overline{n}_{\text{F64}}$$
$$\mid \quad e\ \$\text{x} = \mid \text{x} \mid e\ @ \mid e\ \text{unpack}$$

$$prog ::= e^* \mid \tau^* \mid prog\ prog \mid \varepsilon$$

```
"../stdlib/prelude.phs" require use

(rec:
    $arg =
    (: true ) (: 1 ) $branch fun
    (: arg 0 > ) (: arg 1 - fac arg * ) $branch fun
    branch
) $fac =

(I32) (: fac ) "fac" export
```

```
(func (;1;) (type 0) (param i32) (result i32)
  local.get 0
  i32.const 0
  i32.gt_s
  if (result i32)  ;; label = @1
    local.get 0
    i32.const 1
    i32.sub
    call 1
    local.get 0
    i32.mul
  else
    i32.const 1
  end)
```

```
((I32 (I32) (I32) Arrow):
    $rec =
    $arg =
    arg 0 >
        (): arg -1 + rec @ arg * )
        (): 1 )
    select @
) fix $fac =

10 fac @
```

```
1   "../stdlib/prelude.phs" require use
2
3   (: ( $cond $t $f ) =
4       (: true ) (: f ) $b fun
5       (: cond ) (: t ) $b fun
6       b
7   ) $select =
8
9   (rec:
10      $arg =
11      arg 0 >
12          (: arg 1 - fac arg * )
13          1
14          select
15  ) $fac =
16
17  10 fac :data
18
```

```
Term
Fil  Rediger  Vis  Termi
:data - 3628800n
parse: 76.3503739982843
compile: 241.6710789948
[tate@archbook postfix-
```

$$\frac{e_1 \mapsto e_1' \qquad v_i \text{ val } \forall\ v_i \in v^*}{v^*\ e_1\ \ldots\ e_n \mapsto v^*\ e_1'\ \ldots\ e_n}\ (\text{STEP}) \qquad \frac{\forall\ \tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\overline{n}_\tau \text{ val}}\ (\text{V-1})$$

$$\frac{e^* \mapsto e^{*\prime}}{(e^*) \mapsto (e^{*\prime})}\ (\text{TUPLES-1}) \quad \frac{v_i \text{ val } \forall v_i \in v^*}{(v^*) \text{ val}}\ (\text{TUPLES-2})$$

$$\frac{v_i \text{ val } \forall\ v_i \in v^*}{(v^*)\ \text{unpack} \mapsto v^*}\ (\text{TUPLES-3})$$

$$\frac{v_1 \text{ val} \qquad v_2 \text{ val} \qquad v_3 = v_1 + v_2}{v_1\ v_2\ + \mapsto v_3}\ (\text{ADD})$$

$$\frac{v_1 \text{ val} \qquad v_2 \text{ val} \qquad v_3 = v_1 * v_2}{v_1\ v_2\ \times \mapsto v_3}\ (\text{MUL})$$

$$\frac{v_1 \text{ val} \qquad v_2 \text{ val}}{v_1\ v_2\ == \mapsto \overline{(\delta_{v_1 v_2})}_{\text{I32}}}\ (\text{EQUIV})$$

$$\frac{v_1 \text{ val} \qquad v_2 \text{ val}}{\overline{0}_{\text{I32}}\ v_1\ v_2\ \text{select} \mapsto v_2}\ (\text{TERN-1})$$

$$\frac{v_1 \text{ val} \qquad v_2 \text{ val} \qquad n \neq 0}{\overline{n}_{\text{I32}}\ v_1\ v_2\ \text{select} \mapsto v_1}\ (\text{TERN-2})$$

$$\frac{v \text{ val} \qquad \sigma = [[...], ...[x_1 \mapsto v_1 \ ... \ x_n \mapsto v_n]] \qquad \sigma' = [[...], ...[x_1 \mapsto v_1 \ ... \ x_n \mapsto v_n, x_{n+1} \mapsto v]]}{\sigma; v \ \$\mathtt{x} =\!\mapsto \sigma'; \varepsilon} \ (\text{LET})$$

$$\frac{\mathtt{x} \mapsto v \in \sigma(-1)}{\sigma; \mathtt{x} \mapsto \sigma; v} \ (\textsc{Identifier}) \qquad \frac{\sigma' = \sigma \cup [\,] \qquad v = ((\tau^*) : e^* \,) \qquad v \text{ val}}{\sigma; v \ @ \mapsto \sigma'; e^* \text{end\_scope}} \ (\text{CALL-1})$$

$$\frac{\sigma = [s_1, ...s_n, s_{n+1}] \qquad \sigma' = [s_1, ...s_n]}{\sigma; \text{end\_scope} \mapsto \sigma'; \varepsilon} \ (\text{CALL-2})$$

$$\frac{}{\mathsf{I32\ prim}}\ (\text{P-I32}) \qquad \frac{}{\mathsf{I64\ prim}}\ (\text{P-I64}) \qquad \frac{}{\mathsf{F32\ prim}}\ (\text{P-F32}) \qquad \frac{}{\mathsf{F64\ prim}}\ (\text{P-F64})$$

$$\frac{\tau_1\ \mathsf{prim} \qquad \tau_2\ \mathsf{prim}}{\tau_1\ \tau_2\ \mathsf{Arrow\ prim}}\ (\text{P-ARROW})$$

$$\frac{\tau \rightarrowtail \tau' \qquad \tau_i\ \mathsf{prim}\ \forall\ \tau_i \in \tau_1^*}{\tau_1^*\ \tau\ \tau_2^* \rightarrowtail \tau_1^*\ \tau'\ \tau_2^*}\ (\text{TSTEP}) \qquad \frac{\tau^* \rightarrowtail \tau^{*\prime} \qquad v_i\ \mathsf{val}\ \forall v_i \in v^*}{v^*\ \tau^*\ e^* \rightarrowtail v^*\ \tau^{*\prime}\ e^*}\ (\text{TSTEP - MIXED})$$

$$\frac{e^* \mapsto e^{*\prime} \qquad \tau_i\ \mathsf{prim}\ \forall\ \tau_i \in \tau^*}{\tau^*\ e^*\ \tau_2^* \mapsto \tau^*\ e^{*\prime}\ \tau_1}\ (\text{STEP - MIXED})$$

$$\frac{\tau^* \rightarrowtail \tau^{*\prime}}{(\tau^*) \rightarrowtail (\tau^{*\prime})}\ (\textsc{Tuple Types - 1})\qquad \frac{\tau_i\ \mathsf{prim}\ \forall \tau_i \in \tau^*}{(\tau^*)\ \mathsf{prim}}\ (\textsc{Tuple Types - 2})$$

$$\frac{\tau_i\ \mathsf{prim}\ \forall\ \tau_i \in \tau^*}{(\tau^*)\ \mathsf{unpack} \rightarrowtail \tau_1 ... \tau_n}\ (\textsc{Tuple Types - 3})$$

$$\frac{\sigma = [[...], ...[T_1 \rightarrowtail \tau_1 ... T_n \rightarrowtail \tau_n]] \qquad \begin{array}{c}\tau \text{ prim} \\ \sigma' = [[...], ...[T_1 \rightarrowtail \tau_1 ... T_n \rightarrowtail \tau_n, T_{n+1} \rightarrowtail \tau]]\end{array}}{\sigma; \tau \ \$\mathtt{T} =\!\!\rightarrowtail \sigma'; \varepsilon} \ (\textsc{Type-Let})$$

$$\frac{\sigma = [..., [x_1 \mapsto v_1 \ ... \ x_n \mapsto v_n, \ T_1 \rightarrowtail \tau_1 \ ... \ T_n \rightarrowtail \tau_n]] \qquad \tau_i \text{ val } \forall \ \tau_i \in \tau^*}{\sigma; (\!(\tau^*) : \ e^* \ )\!) \mapsto \sigma; (\!(\tau^*) : \ [v_1/x_1]...[v_n/x_n][\tau_1/T_1]...[\tau_n/T_n]e^* \ )\!) \text{ val}} \ (\text{MACRO-2})$$

$$\frac{\mathtt{T} \mapsto \tau \in \sigma(-1)}{\sigma; \mathtt{T} \rightarrowtail \sigma; \tau} \ (\textsc{Type-Id})$$

$$\frac{\tau_1 \text{ prim} \qquad \tau_2 \text{ prim}}{\tau_1\ \tau_2\ ==\rightarrowtail\ \overline{(\delta_{\tau_1\tau_2})}_{\mathsf{I32}}}\ (\textsc{Type Equiv})$$

$$\frac{((\tau^*\ \tau_1)\colon e^*\ )\ \text{val} \qquad \tau_1 \text{ prim}}{((\tau^*\ \tau_1)\colon e^*\ )\ \tau_1\ \text{fix val}}\ (\textsc{Fixed Point})$$

$$\frac{v = ((\tau^*\ \tau_1)\colon e^*\ )\ \tau_1\ \text{fix} \qquad v\ \text{val}}{v\ @ \mapsto v\ ((\tau^*\ \tau_1)\colon e^*\ )\ @}\ (\textsc{Call FP})$$

$$\frac{\tau^* \rightarrowtail \tau^{*\prime}}{\sigma; ((\tau^*)\colon e^*\ ) \mapsto \sigma; ((\tau^{*\prime})\colon e^*\ )}\ (\textsc{Macro Type Step})$$

$$\frac{v\ \text{val} \qquad v : \tau}{v\ \text{type} \rightarrowtail \tau}\ (\text{TYPE-1})$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \rightarrowtail \tau'}{\Gamma \vdash e : \tau'} \text{ (T-0)}$$

7.2 Math and Logic

$$\frac{}{\Gamma \vdash \overline{n}_{\mathsf{I32}} : \mathsf{I32}} \text{ (T-1)} \qquad \frac{}{\Gamma \vdash \overline{n}_{\mathsf{I64}} : \mathsf{I64}} \text{ (T-2)} \qquad \frac{}{\Gamma \vdash \overline{n}_{\mathsf{F32}} : \mathsf{F32}} \text{ (T-3)} \qquad \frac{}{\Gamma \vdash \overline{n}_{\mathsf{F64}} : \mathsf{F64}} \text{ (T-4)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\Gamma \vdash e_1\ e_2\ + : \tau} \text{ (T-5)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\Gamma \vdash e_1\ e_2\ \times : \tau} \text{ (T-6)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{I32} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ e_2\ e_3\ \mathsf{select} : \tau} \text{ (T-7)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\mathsf{I32}, \mathsf{I64}, \mathsf{F32}, \mathsf{F64}\}}{\Gamma \vdash e_1\ e_2\ == : \mathsf{I32}} \text{ (T-8)} \qquad \frac{\tau_1\ \mathsf{prim} \qquad \tau_2\ \mathsf{prim}}{\Gamma \vdash \tau_1\ \tau_2\ == : \mathsf{I32}} \text{ (T-9)}$$

7.3 Tuples

$$\frac{}{\Gamma \vdash () : \mathsf{Unit}} \text{ (Empty Tuple)} \qquad \frac{\tau^* = \{\tau_i \text{ such that } \Gamma \vdash e_i : \tau_i\ \forall\ e_i \in e^*\}}{\Gamma \vdash (e^*) : (\tau^*)} \text{ (Tuple Type)}$$

$$\frac{\Gamma \vdash (e^*) : (\tau^*)}{\Gamma \vdash (e^*)\ \mathsf{unpack} : \tau^*} \text{ (Unpack)}$$

8.1 Theorem - Progress

If $e^* : \tau^*$ then either $e^* \mapsto e^{*\prime}$ or e_i val $\forall$ $e_i \in e^*$.

Proof. By induction on the derivation of $e^* : \tau^*$.

8.2 Theorem - Preservation

If $\Gamma \vdash e^* : \tau^*$ and $e^* \mapsto e^{*\prime}$ then $\Gamma \vdash e^{*\prime} : \tau^{*\prime}$ where $\tau^{*\prime} \rightarrowtail^* \tau^*$

7.4 Macros

This is why we needed the step judgement

$$\frac{\tau^* = \tau_1 \ \ldots \ \tau_n \qquad v^* = v_1 : \tau_1 \ \ldots \ v_n : \tau_n \qquad m = (\!(\tau^*)\!) : e^* \ ) \qquad m \ \mathsf{val}}{\Gamma \vdash (\!(\tau^*)\!) : e^* \ ) : (\tau^*) \ (v^* \ m \ @) \ \mathsf{type} \ \mathsf{Arrow}} \ (\textsc{Macro})$$

$$\frac{\Gamma \vdash e_1 : \tau^* \ \tau^{*\prime} \ \mathsf{Arrow} \qquad \Gamma \vdash e^* : \tau^*}{\Gamma \vdash e^* \ e_1 \ @ : \tau^{*\prime}} \ (\textsc{Call}) \qquad \frac{\Gamma(\mathbf{x}) = \tau}{\Gamma \vdash \mathbf{x} : \tau} \ (\textsc{Identifier})$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma(\mathbf{x}) = \tau}{\Gamma \vdash e \ \mathbf{x} =: \varepsilon} \ (\textsc{Let})$$

$$\frac{\tau_1 \ \mathsf{prim} \qquad \tau_1 = (\tau_3^*) \ (\tau_4^*) \ \mathsf{Arrow} \qquad \Gamma \vdash e : (\tau_3^* \ \tau_1) \ (\tau_4^*) \ \mathsf{Arrow}}{\Gamma \vdash e \ \tau_1 \ \mathsf{fix} : \tau_1} \ (\textsc{Fix})$$

Conclusion - tis bronk

```
# Some type defs
I32 I64 | $Int =
(Int Int) (Int) Arrow $BinaryOperator =

# FIXME
((Int Int):
    # This doesn't branch properly
    (: ( $a $b ) =
        a type I32 ==
        b type I32 == &&
    ) (: "i32.or" asm ) $or fun
    (: ( $a $b ) =
        a type I64 ==
        b type I64 == &&
    ) (: "i64.or" asm ) $or fun
    or
) type BinaryOperator == :data

# FIXME: classes with unions allow operations on disparate types
```

```
19  ((Int Int):
20      # This doesn't branch properly
21      ((I32 I32): 1 ) ((I32 I32): "i32.or" asm ) $or fun
22      ((I64 I64): 1 ) ((I64 I64): "i64.or" asm ) $or fun
23      or
24  ) type BinaryOperator == :data
25
26  # FIXME: classes with unions allow operations on disparate types
```