An Intuition for Propagators

George Wilson

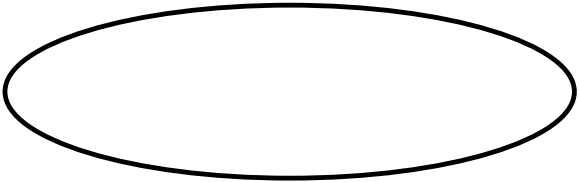
CSIRO's Data61

george.wilson@data61.csiro.au

2nd September 2019

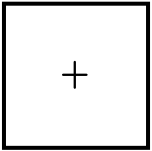
1970s, MIT

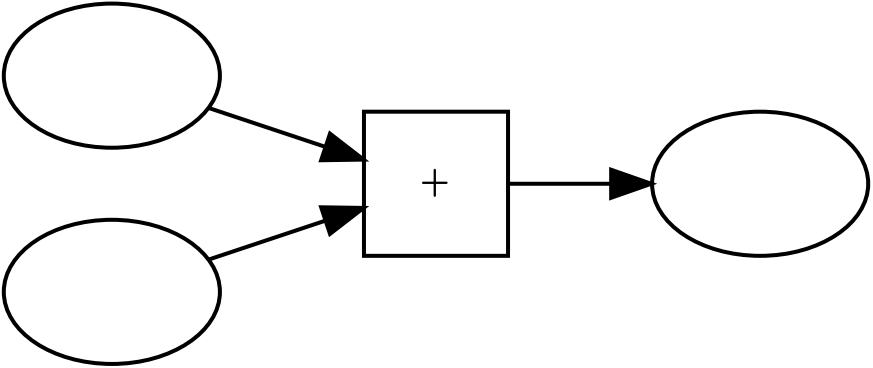
a model of computation for **highly concurrent** machines

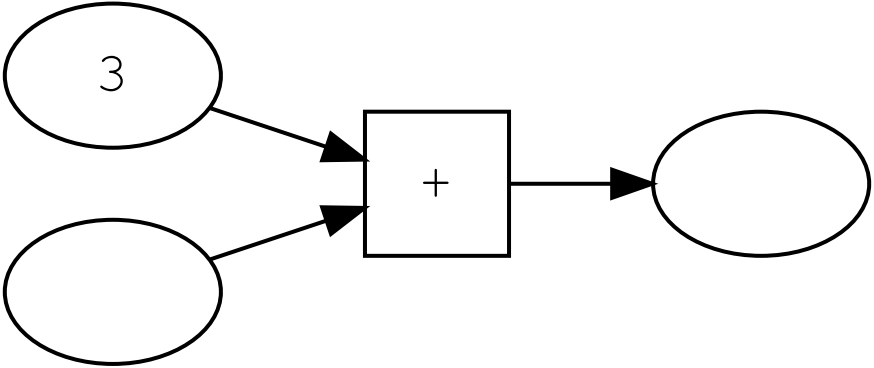


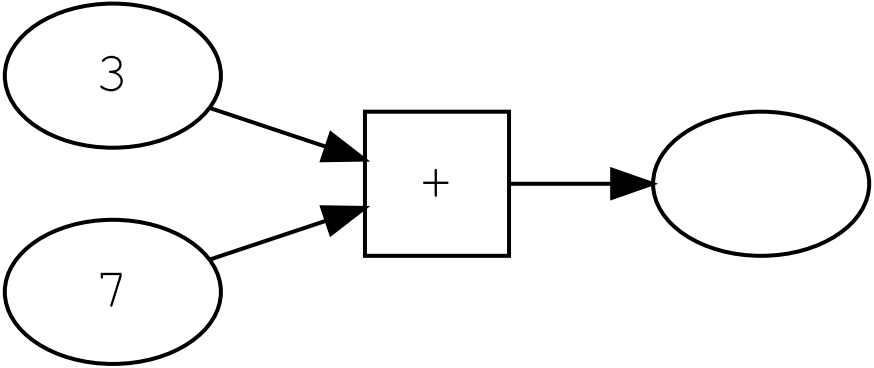
"Hello"

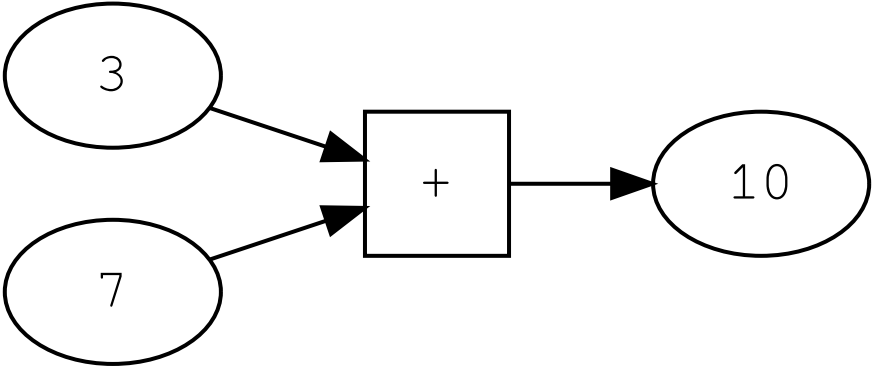
"Compose"

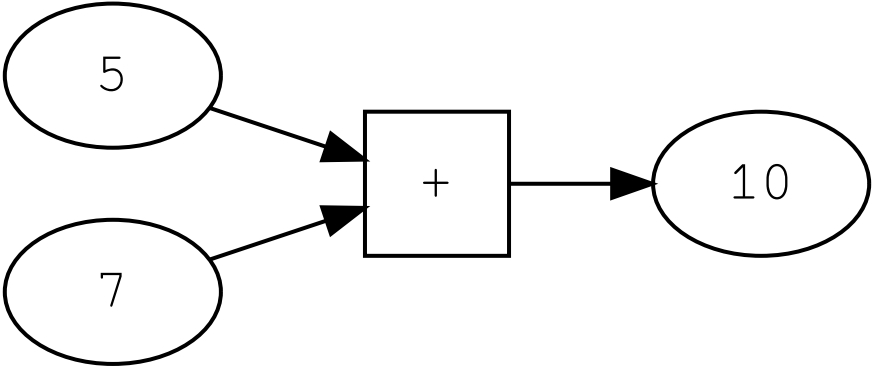


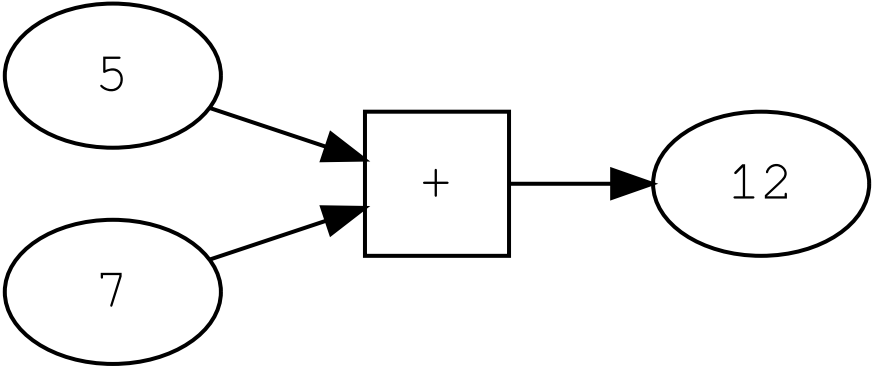



3
+


3
7
+


3
7
+
10


5
7
+
10

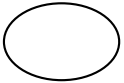

5
7
+
12

```haskell
-- types
data Cell a

data Par a
instance Monad Par
```

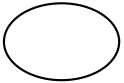
```haskell
-- types
data Cell a

data Par a
instance Monad Par


-- Creating a cell
cell    :: Par (Cell a)
```

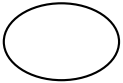
```haskell
-- types
data Cell a

data Par a
instance Monad Par


-- Creating a cell
cell    :: Par (Cell a)


-- Working with Cells
content :: Cell a -> Par (Maybe a)
write   :: Cell a -> a -> Par ()
```

```haskell
-- types
data Cell a

data Par a
instance Monad Par


-- Creating a cell
cell    :: Par (Cell a)


-- Working with Cells
content :: Cell a -> Par (Maybe a)
write   :: Cell a -> a -> Par ()


-- Creating a propagator
watch   :: Cell a -> (a -> Par ()) -> Par ()
```

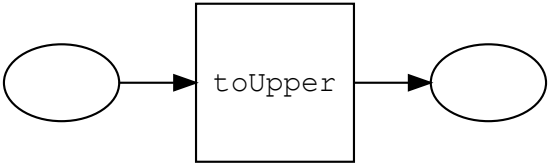


```
do
  input  <- cell
```

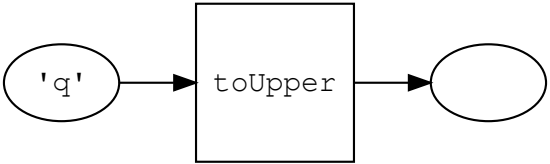




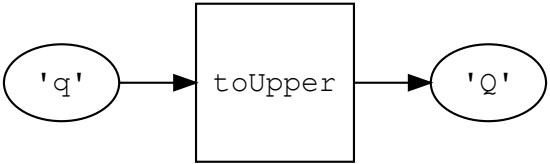
```
do
  input  <- cell
  output <- cell
```



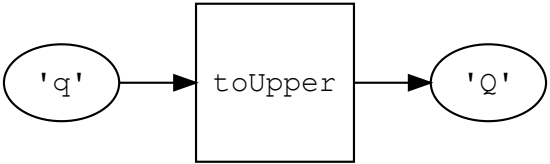

```
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))
```




```haskell
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))

  write input 'q'
  content output   -- Just 'Q'
```

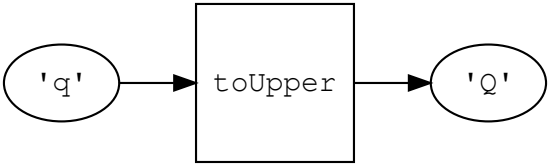



```haskell
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))

  write input 'q'
  content output   -- Just 'Q'
```



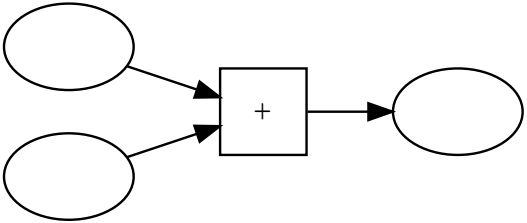

```haskell
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))

  write input 'q'
  content output    -- Just 'Q'
```

```haskell
lift :: (a -> b) -> Cell a -> Cell b -> Par ()
lift f input output =
  watch input (\a ->
    write output (f a))
```

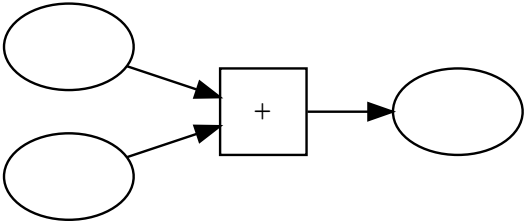



```haskell
do
  input  <- cell
  output <- cell
  lift toUpper input output

  write input 'q'
  content output    -- Just 'Q'
```

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell
```

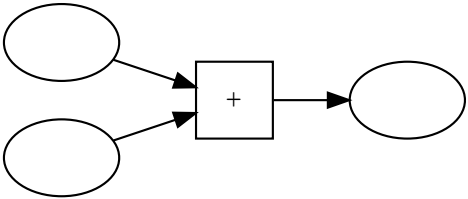



```haskell
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  watch inL (\x -> do
    maybeY <- content inR
    case maybeY of
      Nothing -> pure ()
      Just y  -> write out (x+y)
```

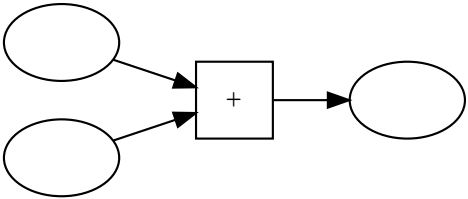



```haskell
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  watch inL (\x -> do
    maybeY <- content inR
    case maybeY of
      Nothing -> pure ()
      Just y  -> write out (x+y)

  watch inR (\y -> do
    maybeX <- content inL
    case maybeX of
      Nothing -> pure ()
      Just x  -> write out (x+y)
```

```haskell
with :: Cell a -> (a -> Par ()) -> Par ()
with theCell callback = do
  maybeA <- content theCell
    case maybeA of
      Nothing -> pure ()
      Just a  -> callback a
```



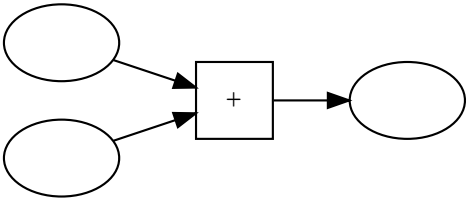

```haskell
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  watch inL (\x ->
    with inR (\y ->
      write out (x+y)
```




```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  watch inL (\x ->
    with inR (\y ->
      write out (x+y)

  watch inR (\y ->
    with inL (\x ->
      write out (x+y)
```

```
lift2 :: (a -> b -> c)
      -> Cell a -> Cell b -> Cell c
      -> Par ()
```

```haskell
lift2 :: (a -> b -> c)
      -> Cell a -> Cell b -> Cell c
      -> Par ()
lift2 f inL inR out = do
  watch inL (\a ->
    with inR (\b ->
      write out (f a b)))
  watch inR (\b ->
    with inL (\a ->
      write out (f a b)))
```




```haskell
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  adder inL inR out

    where
      adder l r o = do
        lift2 (+) l r o
        lift2 (-) o l r
        lift2 (-) o r l
```

?!

How can we fix this?

```haskell
data WriteOnce a
  = None
  | Written a
  | TooMany

write ::            a -> WriteOnce a -> WriteOnce a
write a w = case w of
  None      -> Written a
  Written b -> TooMany
  TooMany   -> TooMany
```

```haskell
data WriteOnce a
  = None
  | Written a
  | TooMany

write :: (Eq a) => a -> WriteOnce a -> WriteOnce a
write a w = case w of
  None       -> Written a
  Written b -> if a == b then Written b else TooMany
  TooMany    -> TooMany
```

Accumulate information about a value

Accumulate information about a value

```haskell
data WriteOnce a

  -- I don't know anything
  = None

  -- I know the answer exactly
  | Written a

  -- I have heard contradictory answers!
  | TooMany
```

Accumulate information about a value

The accumulation must:

- tolerate **reordering** of information
- tolerate **grouping** of information
- ignore **redencancy** of information

Bounded join semilattice

Identity:

$$x \vee bottom = bottom = bottom \vee x$$

Commutative:

$$x \vee y = y \vee x$$

Associative:

$$x \vee (y \vee z) = (x \vee y) \vee z$$

Idempotent:

$$x \vee x = x$$

```haskell
class SemiLattice a where
  (\/)   :: a -> a -> a
  bottom :: a

instance (Eq a) => SemiLattice (WriteOnce a) where
  None       \/ b            = b
  TooMany    \/ x            = TooMany
  Written a \/ None          = Written a
  Written a \/ Written b     = Written a
  Written a \/ TooMany       = TooMany
```

```haskell
instance (Ord a) => Semilattice (Set a) where
  p \/ q = Set.union p q
  bottom = Set.empty
```

Monotonicity

$$x \leq y \implies f(x) \leq f(y)$$

Thanks for listening!