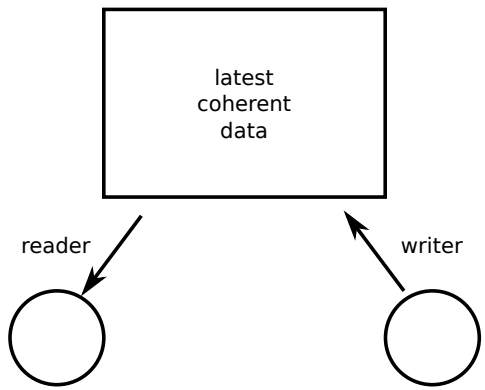
Functional Pearl: Four slot asynchronous communication mechanism

Matthew Danish

September 17, 2013

Introduction

- Low-level systems programming with dependent types
- Simpson, 1989. *Four slot fully asynchronous communication mechanism*.



- No synchronization or delay caused to reader or writer
- Reader sees single piece of coherent data from writer
- Requires a "four slot array" to operate safely

Four slot mechanism: state

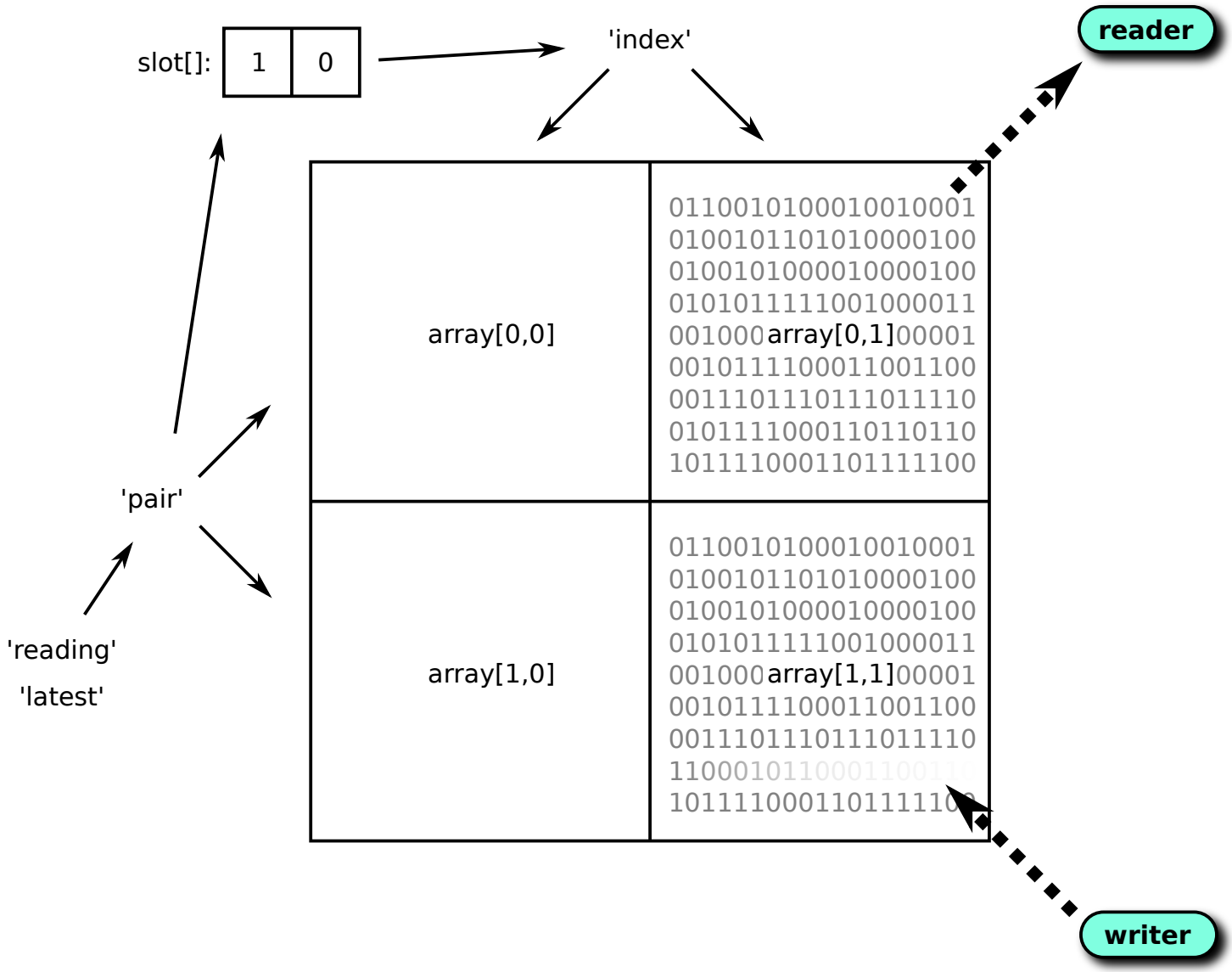
Global state

- The 'reading' variable, $R : bit$.
- The 'latest' variable, $L : bit$.
- The 2-slot bit array of indices, $slot : \{bit, bit\}$.
- The 4-slot array of data, $array : \{\{\alpha, \alpha\}, \{\alpha, \alpha\}\}$.

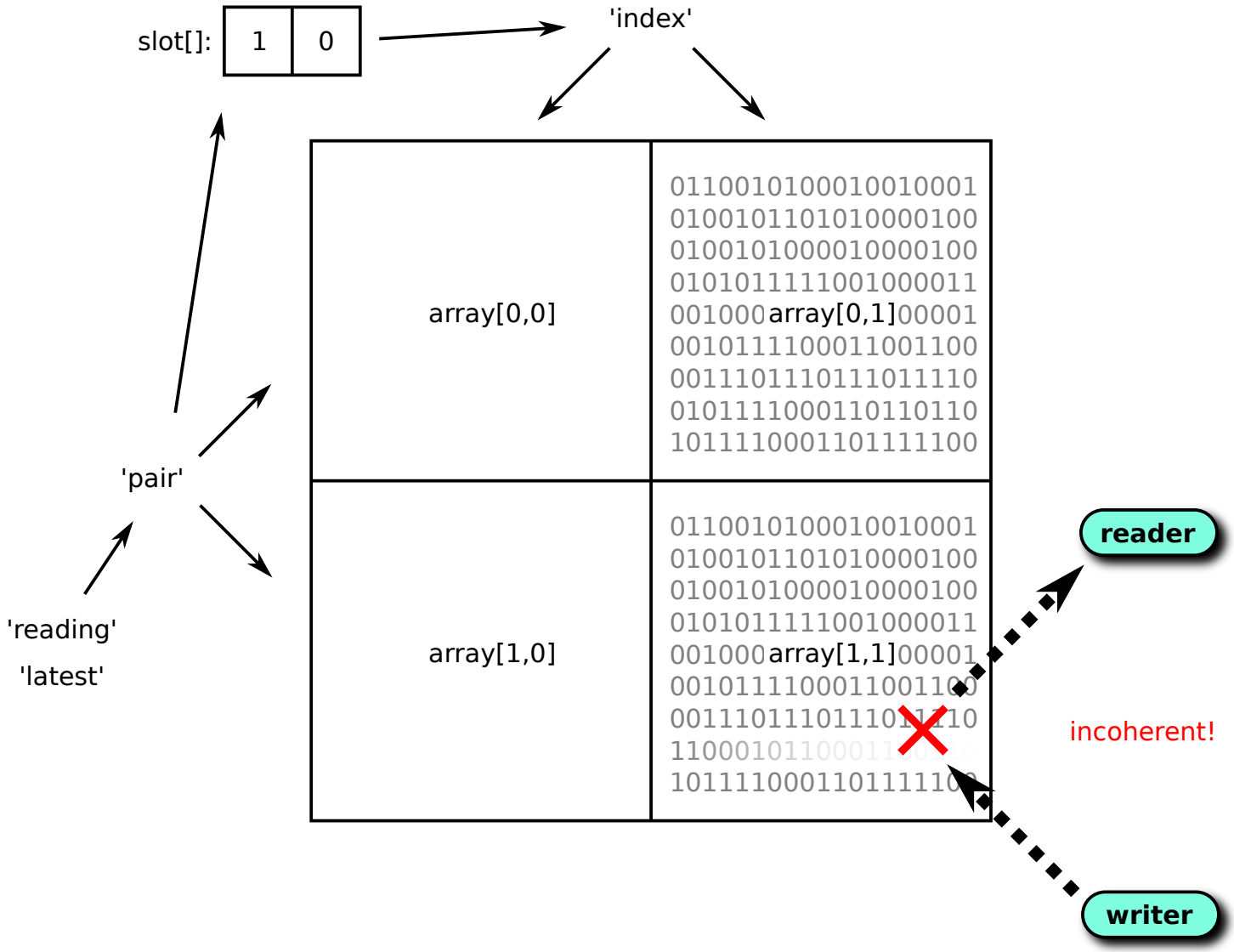
Local state

- The 'pair' chosen by writer or reader $w_p, r_p : bit$.
- The 'index' chosen by writer or reader $w_i, r_i : bit$.

Four slot mechanism: dataflow



Four slot mechanism: incoherence



Four slot mechanism

Writer

WS1 $w_p \leftarrow \neg R$

WS2 $w_i \leftarrow \neg slot\,[w_p]$

WS3 $\texttt{write\_data}\,(w_p, w_i, item)$

WS4 $slot\,[w_p] \leftarrow w_i$

WS5 $L \leftarrow w_p$

Four slot mechanism

Writer

WS1 $w_p \leftarrow \neg R$

WS2 $w_i \leftarrow \neg slot\,[w_p]$

WS3 $\texttt{write\_data}\,(w_p, w_i, item)$

WS4 $slot\,[w_p] \leftarrow w_i$

WS5 $L \leftarrow w_p$

Reader

RS1 $r_p \leftarrow L$

RS2 $R \leftarrow r_p$

RS3 $r_i \leftarrow slot\,[r_p]$

RS4 $item \leftarrow \texttt{read\_data}\,(r_p, r_i)$

RS5 return $item$

Arbitrary interleaving

- Suppose $L = 1$ and $R = 0$

WS1 $w_p \leftarrow \neg R$

RS1 $r_p \leftarrow L$

Arbitrary interleaving

- Suppose $L = 1$ and $R = 0$

WS1 $w_p \leftarrow \neg R$

RS1 $r_p \leftarrow L$

- Now $w_p = r_p$

Arbitrary interleaving

- Suppose $L = 1$ and $R = 0$

WS1 $w_p \leftarrow \neg R$

RS1 $r_p \leftarrow L$

- Now $w_p = r_p$

WS2 $w_i \leftarrow \neg slot\,[w_p]$

RS2 $R \leftarrow r_p$

RS3 $r_i \leftarrow slot\,[r_p]$

- And $w_i \neq r_i$

Arbitrary interleaving

- Suppose $L = 1$ and $R = 0$

WS1 $w_p \leftarrow \neg R$

RS1 $r_p \leftarrow L$

- Now $w_p = r_p$

WS2 $w_i \leftarrow \neg slot\,[w_p]$

RS2 $R \leftarrow r_p$

RS3 $r_i \leftarrow slot\,[r_p]$

- And $w_i \neq r_i$

WS3 $\texttt{write\_data}\,(w_p, w_i, item)$

RS4 $item \leftarrow \texttt{read\_data}\,(r_p, r_i)$

- ...

Coherency property

Theorem (Coherency)

The writer and the reader do not access the same data slot at the same time. More precisely, this assertion must be satisfied at potentially conflicting program points WS3 *and* RS4*:*

$$w_p \neq r_p \vee w_i \neq r_i$$

Coherency property

Theorem (Coherency)

The writer and the reader do not access the same data slot at the same time. More precisely, this assertion must be satisfied at potentially conflicting program points WS3 *and* RS4*:*

$$w_p \neq r_p \vee w_i \neq r_i$$

Problem:
w_p and r_p (w_i and r_i) are local variables in separate processes

Static dependent types to the rescue!

- Observed values of atomic variables $R, L, slot\,[]$ can tell us facts about unseen state, for instance:

Static dependent types to the rescue!

- Observed values of atomic variables $R, L, slot\ []$ can tell us facts about unseen state, for instance:
- $$\left.\begin{array}{ll} \text{RS2} & R \leftarrow r_p \\ \text{WS1} & w_p \leftarrow \neg R \end{array}\right\} w_p \neq r_p \text{ at WS1}$$

Static dependent types to the rescue!

- Observed values of atomic variables $R, L, slot\ []$ can tell us facts about unseen state, for instance:
- $\left.\begin{array}{ll} \text{RS2} & R \leftarrow r_p \\ \text{WS1} & w_p \leftarrow \neg R \end{array}\right\} w_p \neq r_p$ at WS1
- $\left.\begin{array}{ll} \text{WS1} & w_p \leftarrow \neg R \\ \text{RS2} & R \leftarrow r_p \end{array}\right\} w_p \stackrel{?}{=} r_p$ at WS1

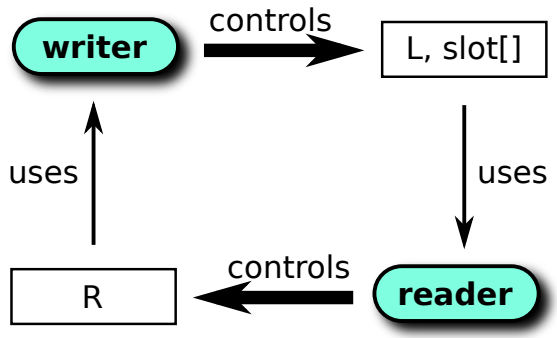
Property (Interaction of WS1 and RS2)

If $w_p = r_p$ *at* WS1 *then* WS1 *preceded* RS2*.*

Static dependent types to the rescue!

Theorem

If WS1 *precedes* RS2 *then it also precedes* RS3 $r_i \leftarrow slot\,[r_p]$.



- The writer controls the values of *slot* [] and L
- The reader has only one choice for r_p, r_i.
- Therefore, the writer merely needs to pick the opposite index.
- Let's encode these kind of properties into types.

WS1

$$w_p \leftarrow \neg R$$

```
absview ws1_read_v (R: bit, rstep: int, rp: bit)

fun get_reading_state ():
  [rstep: nat]
  [R, rp: bit | R == rp || (R <> rp ==> rstep < 2)]
  (ws1_read_v (R, rstep, rp) | bit R)
```

WS2

$$w_i \leftarrow \neg slot\,[w_p]$$

```
absview ws2_slot_v (s: bit, rp: bit, ri: bit)

fun get_write_slot_index {R, wp, rp: bit} {rstep: nat} (
    pfr: !ws1_read_v (R, rstep, rp) |
    wp: bit wp
  ): [s, ri: bit | (rstep < 3 && wp == rp) ==> s == ri)]
     (ws2_slot_v (s, rp, ri) | bit s)
```

WS3

$$\texttt{write\_data}\,(w_p, w_i, \mathit{item})$$

```
fun{a: t@ype} write_data
  {R, s, wp, wi, rp, ri: bit | wp <> rp || wi <> ri} {rstep: nat} (
    pfr: !ws1_read_v (R, rstep, rp),
    pfs: !ws2_slot_v (s, rp, ri) |
    wp: bit wp, wi: bit wi, item: a
  ): void
```

WS4

$$slot\,[w_p] \leftarrow w_i$$

```
absview ws4_fresh_v (p: bit)

fun save_write_slot_index
  {R, s, wp, wi, rp, ri: bit | wi <> s} {rstep: nat} (
    pfr: !ws1_read_v (R, rstep, rp),
    pfs: ws2_slot_v (s, rp, ri) |
    wp: bit wp, wi: bit wi
  ): (ws4_fresh_v wp | void)
```

WS5

$$L \leftarrow w_p$$

```
fun save_latest_state
  {R, rp, wp: bit | wp <> R} {rstep: nat} (
    pfr: ws1_read_v (R, rstep, rp),
    pff: ws4_fresh_v wp |
    wp: bit wp
  ): void
```

write

```
(* Step 1 *)
val (pfr | R) = get_reading_state ()
val wp = not R

(* Step 2 *)
val (pfs | s) = get_write_slot_index (pfr | wp)
val wi = not s

(* Step 3 *)
val _ = write_data (pfr, pfs | wp, wi, item)

(* Step 4 *)
val (pff | _) = save_write_slot_index (pfr, pfs | wp, wi)

(* Step 5 *)
val _ = save_latest_state (pfr, pff | wp)
```

WS1 $w_p \leftarrow \neg R$

WS2 $w_i \leftarrow \neg slot\left[w_p\right]$

WS3 $\texttt{write\_data}\left(w_p, w_i, item\right)$

WS4 $slot\left[w_p\right] \leftarrow w_i$

WS5 $L \leftarrow w_p$

Conclusion

- No overhead: Types erased during compilation.
- Each step compiles to a line or two of C code.
- Dependent types mixed with systems programming.
- Stronger specifications, more confidence, fewer bugs.