Ready, set, Go!

Data-race detection and the Go language

Daniel S. Fava
`danielsf@ifi.uio.no`

Martin Steffen
`msteffen@ifi.uio.no`

Department of informatics
University of Oslo, Norway

SBMF'19
São Paulo, Brazil

Tools that detect data races are important.

Tools that detect data races are important.

Data races

Tools that detect data races are important.

Data races

- Subtle bugs

Tools that detect data races are important.

Data races

- Subtle bugs
- Unknown semantics (under weak-memory)

A memory model informs us
how our multi-threaded programs behave.

A memory model informs us how our multi-threaded programs behave.

Initially `z = 0; done = false;`

```
     T1                     |        T2
z    = 42                   |
done = true                 |   while (!done) {}
                            |   print("t2", z)
```

Relaxed memory models are complex.

The DRF-SC guarantee helps programmers.

If program is Data-Race Free (DRF) then
memory behaves Sequentially Consistently (SC).

The DRF-SC guarantee helps programmers.

If program is Data-Race Free (DRF) then
memory behaves Sequentially Consistently (SC).

P P

The DRF-SC guarantee helps programmers.

If program is Data-Race Free (DRF) then
memory behaves Sequentially Consistently (SC).

P P

The DRF-SC guarantee helps programmers.

If program is Data-Race Free (DRF) then
memory behaves Sequentially Consistently (SC).

$[\![P]\!]_{sc}$ $\qquad\qquad$ P

The DRF-SC guarantee helps programmers.

If program is Data-Race Free (DRF) then
memory behaves Sequentially Consistently (SC).

$[\![P]\!]_{sc}$ $[\![P]\!]_{w}$

The DRF-SC guarantee helps programmers.

If program is Data-Race Free (DRF) then
memory behaves Sequentially Consistently (SC).

$$[\![P]\!]_{sc} \quad \subseteq \quad [\![P]\!]_{w}$$

The DRF-SC guarantee helps programmers.

If program is Data-Race Free (DRF) then
memory behaves Sequentially Consistently (SC).

$$[\![P]\!]_{sc} \subseteq [\![P]\!]_{w}$$

The DRF-SC guarantee helps programmers.

If program is Data-Race Free (DRF) then
memory behaves Sequentially Consistently (SC).

$$[\![P]\!]_{sc} \quad = \quad [\![P]\!]_{w}$$

There is little about data-race detection and channels.
Instead, there has been research on...

There is little about data-race detection and channels. Instead, there has been research on...

✓Race detection & message passing.

There is little about data-race detection and channels. Instead, there has been research on...

✓ Race detection & message passing.

- No shared memory.
- Races as competing to send-to/receive-from channels.
- Absence of races imply determinism.

There is little about data-race detection and channels. Instead, there has been research on...

✓ Race detection & message passing.

- No shared memory.
- Races as competing to send-to/receive-from channels.
- Absence of races imply determinism.

Origin of the *happens-before* relation and *vector clocks.*

There is little about data-race detection and channels. Instead, there has been research on...

There is little about data-race detection and channels. Instead, there has been research on...

✓ Data-race detection & locks

There is little about data-race detection and channels. Instead, there has been research on...

✓ Data-race detection & locks

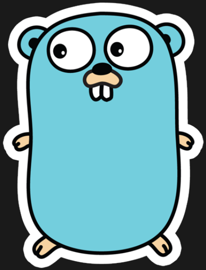
Shared memory, but

no channels,

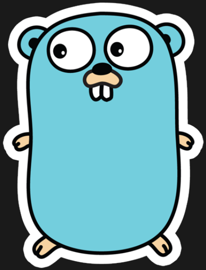
no synchronization via message passing.

We express race-detection for a language with message passing as the sole synchronization primitive.

We express race-detection for a language with message passing as the sole synchronization primitive.



We express race-detection for a language with message passing as the sole synchronization primitive.



Operational Semantics of a Weak Memory Model with Channel Synchronization

Daniel Fava   Martin Steffen   Volker Stolz

[FM'18, JLAMP'18]

II. Background

A *data race* constitutes memory accesses that *conflict* and are *concurrent*.

A *data race* constitutes memory accesses that *conflict* and are *concurrent.*

$$\text{Conflict}\begin{cases}\text{same memory location,}\\ \text{at least one access is a write.}\end{cases}$$

A *data race* constitutes memory accesses that *conflict* and are *concurrent.*

$$\text{Conflict}\ \begin{cases}\text{same memory location,}\\ \text{at least one access is a write.}\end{cases}$$

Concurrent: not ordered by happens-before.

The Go memory model.

[Go memory model, 2014]
replacing *thread* by *goroutine*

The Go memory model.

- *Within a single thread,*
  - *reads and writes must behave as if they executed in the order specified by the program;*

[Go memory model, 2014]
replacing *thread* by *goroutine*

The Go memory model.

- *Within a single thread,*
  - *reads and writes must behave as if they executed in the order specified by the program;*
  - *reorder is allowed only when it does not change the behavior within that thread.*

[Go memory model, 2014]
replacing *thread* by *goroutine*

The Go memory model.

- *Within a single thread,*
  - *reads and writes must behave as if they executed in the order specified by the program;*
  - *reorder is allowed only when it does not change the behavior within that thread.*
- *The execution order observed by one thread may differ from the order observed by another.*

[Go memory model, 2014]
replacing *thread* by *goroutine*

Initially `z = 0; done = false;`

```
     T1                        |        T2
z    = 42                      |
done = true                    |   if (done)
                               |      print("t2", z)
```

Initially `z = 0; done = false;`

```
     T1                    |        T2
z    = 42    (A)           |
done = true  (B)           |   if (done)           (C)
                           |      print("t2", z)  (D)
```

Initially `z = 0; done = false;`

```
     T1                     |       T2
z    = 42    (A)            |
done = true  (B)            |   if (done)            (C)
                            |      print("t2", z)  (D)
```

$A \rightarrow_{hb} B$

Initially `z = 0; done = false;`

```
     T1                       |         T2
z    = 42     (A)             |
done = true   (B)             |   if (done)           (C)
                              |      print("t2", z)   (D)
```

$A \rightarrow_{hb} B$ $\qquad$ $C \rightarrow_{hb} D$

The Go memory model.

[Go memory model, 2014]

The Go memory model.

A *send* happens-before the corresponding *receive* completes.

[Go memory model, 2014]

The Go memory model.

A *send* happens-before the corresponding *receive* completes.

Given a channel c with capacity k,
the i^{th} receive from c happens-before the $(i+k)^{th}$ send completes.

[Go memory model, 2014]

Initially `z = 0; done = false;`

```
     T1                    |        T2
z    = 42    (A)           |
done = true  (B)           |   if (done)            (C)
                           |      print("t2", z)  (D)
```

$A \rightarrow_{hb} B$ $C \rightarrow_{hb} D$

Initially `z = 0; done = false;`

```
      T1                 |        T2
z    = 42     (A)        |
done = true   (B)        |   if (done)            (C)
                         |      print("t2", z)  (D)
```

$\mathtt{A} \rightarrow_{\mathsf{hb}} \mathtt{B}$

$\mathtt{C} \rightarrow_{\mathsf{hb}} \mathtt{D}$

Initially `z = 0; done = false;`

```
      T1                  |        T2
z     = 42    (A)         |
c   <- true   (B)         |   if (done)           (C)
                          |      print("t2", z)   (D)
```

$\texttt{A} \rightarrow_{\mathsf{hb}} \texttt{B}$ $\texttt{C} \rightarrow_{\mathsf{hb}} \texttt{D}$

Initially `z = 0; done = false;`

```
      T1                  |         T2
z     = 42    (A)         |
c   <- true   (B)         |   if (done)           (C)
                          |      print("t2", z)  (D)
```

$\mathtt{A} \rightarrow_{\mathsf{hb}} \mathtt{B}$

$\mathtt{C} \rightarrow_{\mathsf{hb}} \mathtt{D}$

Initially `z = 0; done = false;`

```
     T1                        |         T2
z    = 42    (A)               |
c   <- true  (B)               |   if (done)            (C)
                               |      print("t2", z)  (D)
```

$\texttt{A} \rightarrow_{\mathsf{hb}} \texttt{B}$

$\texttt{C} \rightarrow_{\mathsf{hb}} \texttt{D}$

Initially `z = 0; done = false;`

```
      T1                    |         T2
z     = 42    (A)           |
c   <- true   (B)           |      <- c           (C)
                            |      print("t2", z)  (D)
```

$\mathtt{A} \rightarrow_{\mathsf{hb}} \mathtt{B}$ $\mathtt{C} \rightarrow_{\mathsf{hb}} \mathtt{D}$

Initially `z = 0; done = false;`

```
      T1                    |        T2
z     = 42     (A)          |
c   <- true    (B)          |      <- c              (C)
                            |      print("t2", z)    (D)
```

$A \rightarrow_{hb} B$

$C \rightarrow_{hb} D$

Initially `z = 0; done = false;`

```
     T1                   |       T2
z    = 42    (A)          |
c   <- true  (B)          |     <- c              (C)
                          |     print("t2", z)  (D)
```

$A \rightarrow_{hb} B$ $C \rightarrow_{hb} D$

$B \rightarrow_{hb} C$

Initially `z = 0; done = false;`

```
      T1                           |         T2
z     = 42    (A)                  |
c   <- true   (B)                  |       <- c              (C)
                                   |       print("t2", z)  (D)
```

$A \to_{hb} B$ $\quad$ $C \to_{hb} D$

$B \to_{hb} C$

$$A \to_{hb} D$$

III. Our approach

III. Our approach

Intuition

III. Our approach

Intuition

Efficiency

An access to memory is captured by an event

$(m, ?z) \qquad (m', !z)$

An access to memory is captured by an event

$(m, ?z)$ $(m', !z)$

An access to memory is captured by an event

$(m, ?z)$ $(m', !z)$

An access to memory is captured by an event

$(m, ?z)$ $\quad$ $(m', !z)$

An access to memory is captured by an event

$(m, ?z)$ $(m', !z)$

An access to memory is captured by an event

$(m, ?z)$ $\quad$ $(m', !z)$

An access to memory is captured by an event

$(m, ?z)$ $(m', !z)$

Each thread keeps track of events in its past

- Happened-before set, E_{hb}

An access to memory is captured by an event

$(m, ?z)$ $(m', !z)$

Each thread keeps track of events in its past

- Happened-before set, E_{hb}

A memory cell keeps track of

- a variable's value
- set of events that have happened-before to the variable, E_{hb}

$p\langle E_{hb}, t\rangle$

$p\langle E_{hb}, t\rangle$ $\qquad$ $(\!|E^{z}_{hb},\ z{:=}v|\!)$

$$p\langle E_{hb}, t\rangle \parallel (\!| E^z_{hb},\ z{:=}v |\!)$$

$$p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!)$$

$$
\begin{aligned}
& p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\
& \quad \rightarrow \quad p\langle E'_{hb}, t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v' |\!)
\end{aligned}
$$

$$\frac{}{\begin{array}{l} p\langle E_{hb}, z := v'; t\rangle \parallel (\!|E^{z}_{hb},\ z{:=}v|\!) \\ \quad \rightarrow \quad p\langle E'_{hb}, t\rangle \parallel (\!|E'^{z}_{hb},\ z{:=}v'|\!) \end{array}}$$

$$\frac{\mathit{fresh}(m)}{\begin{array}{c} p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \quad\rightarrow\quad p\langle E'_{hb}, t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v' |\!) \end{array}}$$

$$\frac{\textit{fresh}(m) \qquad \color{red}{(m, !z)}}{\begin{array}{c} p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \rightarrow \quad p\langle E'_{hb}, t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v' |\!) \end{array}}$$

$$\frac{fresh(m) \qquad E'_{hb} = \{(m, !z)\} \cup E_{hb}}{\begin{array}{l} p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \quad \rightarrow \quad p\langle E'_{hb}, t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v' |\!) \end{array}}$$

$$\frac{fresh(m) \qquad \begin{array}{l} E'_{hb} = \{(m, !z)\} \cup E_{hb} \\ E'^{z}_{hb} = \{(m, !z)\} \cup E^{z}_{hb} \end{array}}{\begin{array}{l} p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \quad \rightarrow \quad p\langle E'_{hb}, t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v' |\!) \end{array}}$$

A write is allowed to proceed if...

$$\frac{fresh(m) \qquad \begin{array}{l} E'_{hb} = \{(m, !z)\} \cup E_{hb} \\ E'^{z}_{hb} = \{(m, !z)\} \cup E^{z}_{hb} \end{array}}{\begin{array}{l} p\langle E_{hb}, z := v'; t\rangle \parallel (\!|E^{z}_{hb},\ z{:=}v|\!) \\ \quad \rightarrow \quad p\langle E'_{hb}, t\rangle \parallel (\!|E'^{z}_{hb},\ z{:=}v'|\!) \end{array}}$$

A write is allowed to proceed if...

$$\frac{fresh(m) \qquad \begin{array}{c} E'_{hb} = \{(m, !z)\} \cup E_{hb} \\ E'^{z}_{hb} = \{(m, !z)\} \cup E^{z}_{hb} \end{array} \qquad E^{z}_{hb} \subseteq E_{hb}}{\begin{array}{c} p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \rightarrow \quad p\langle E'_{hb}, t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v' |\!) \end{array}}$$

$$p\langle E_{hb}, \texttt{let } r = \texttt{load } z \texttt{ in } t\rangle \parallel (\!| E^{z}_{hb}, z{:=}v |\!)$$

$$
\begin{aligned}
& p\langle E_{hb}, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel (\!| E^z_{hb},\ z{:=}v |\!) \\
\rightarrow \quad & p\langle E'_{hb}, \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle \parallel (\!| E'^z_{hb},\ z{:=}v |\!)
\end{aligned}
$$

$$\frac{}{\begin{array}{l} p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel \llparenthesis E^{z}_{hb},\ z{:=}v \rrparenthesis \\ \quad \to \quad p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel \llparenthesis E'^{z}_{hb},\ z{:=}v \rrparenthesis \end{array}}$$

$$\frac{\mathit{fresh}(m)}{\begin{array}{l} p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \quad \rightarrow \quad p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v |\!) \end{array}}$$

$$\dfrac{\mathit{fresh}(m) \qquad (m, ?z)}{\begin{array}{l} p\langle E_{hb}, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel \llparenthesis E^{z}_{hb},\ z{:=}v \rrparenthesis \\ \quad \rightarrow \quad p\langle E'_{hb}, \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle \parallel \llparenthesis E'^{z}_{hb},\ z{:=}v \rrparenthesis \end{array}}$$

$$\frac{fresh(m) \qquad E'_{hb} = \{(m, ?z)\} \cup E_{hb}}{\begin{array}{l} p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \quad \rightarrow \quad p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v |\!) \end{array}}$$

$$\frac{\mathit{fresh}(m) \qquad \begin{array}{l} E'_{hb} = \{(m, ?z)\} \cup E_{hb} \\ E'^{z}_{hb} = \{(m, ?z)\} \cup E^{z}_{hb} \end{array}}{\begin{array}{l} p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \quad \to \quad p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v |\!) \end{array}}$$

A read is allowed to proceed if...

$$\frac{\mathit{fresh}(m) \qquad \begin{array}{l} E'_{hb} = \{(m, ?z)\} \cup E_{hb} \\ E'^{z}_{hb} = \{(m, ?z)\} \cup E^{z}_{hb} \end{array}}{\begin{array}{l} p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \quad \rightarrow \quad p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v |\!) \end{array}}$$

A read is allowed to proceed if...

$$\frac{\mathit{fresh}(m) \qquad \begin{array}{l} E'_{hb} = \{{\color{cyan}(m, ?z)}\} \cup E_{hb} \\ E'^{z}_{hb} = \{{\color{cyan}(m, ?z)}\} \cup E^{z}_{hb} \end{array} \qquad E^{z}_{hb} \downarrow_! \subseteq E_{hb}}{\begin{array}{c} p\langle E_{hb}, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel (\!| E^{z}_{hb},\ z{:=}v |\!) \\ \rightarrow \quad p\langle E'_{hb}, \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle \parallel (\!| E'^{z}_{hb},\ z{:=}v |\!) \end{array}}$$

Sends and Receives transmit a thread's happens-before set

Sends and Receives transmit a thread's happens-before set

Threads "learn" from each other about past events

Channel receive

$$\frac{v \neq \bot \qquad E'_{hb} = E_{hb} + E''_{hb}}{\begin{array}{rcll} c_b[q_1] \parallel & p\langle E_{hb}, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle & \parallel c_f[q_2 :: (v, E''_{hb})] & \rightarrow \\ c_b[E_{hb} :: q_1] \parallel & p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle & \parallel c_f[q_2] & \end{array}}$$

Channel receive

$$\frac{v \neq \perp \qquad E'_{hb} = E_{hb} + E''_{hb}}{\begin{array}{rcl} c_b[q_1] \parallel & p\langle E_{hb}, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle & \parallel c_f[q_2 :: (v, E''_{hb})] \quad \rightarrow \\ c_b[E_{hb} :: q_1] \parallel & p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle & \parallel c_f[q_2] \end{array}}$$

Channel receive

$$\frac{v \neq \bot \qquad E'_{hb} = E_{hb} + E''_{hb}}{\begin{array}{rcll} c_b[q_1] \parallel & p\langle E_{hb}, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle & \parallel c_f[q_2 :: (v, E''_{hb})] & \rightarrow \\ c_b[E_{hb} :: q_1] \parallel & p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle & \parallel c_f[q_2] & \end{array}}$$

Channel send

$$\frac{\neg closed(c_f[q_2]) \qquad E'_{hb} = E_{hb} + E''_{hb}}{\begin{array}{r}c_b[q_1 :: E''_{hb}] \parallel \quad p\langle E_{hb}, c \leftarrow v; t\rangle \quad \parallel c_f[q_2] \quad \rightarrow \\ c_b[q_1] \parallel \qquad p\langle E'_{hb}, t\rangle \qquad \parallel c_f[(v, E_{hb}) :: q_2]\end{array}}$$

Efficiency


z_0

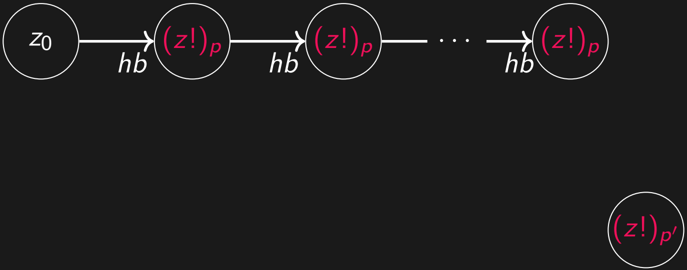
z_0 $(z!)_p$

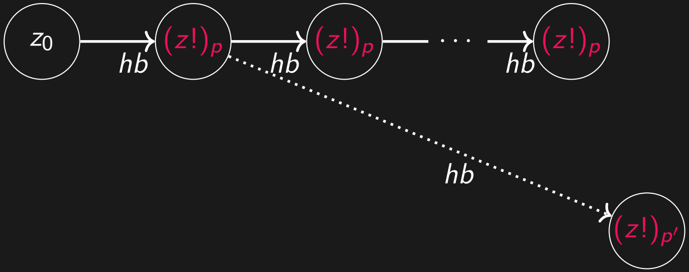
$$z_0 \xrightarrow[hb]{} (z!)_p$$

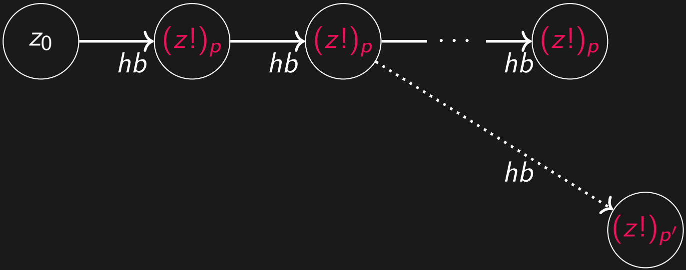
$$z_0 \xrightarrow{hb} (z!)_p \qquad (z!)_p$$

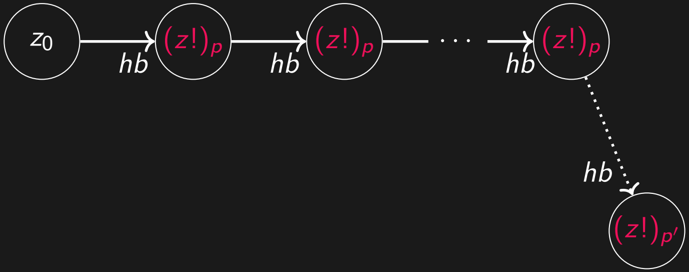
$$z_0 \xrightarrow{hb} (z!)_p \xrightarrow{hb} (z!)_p$$

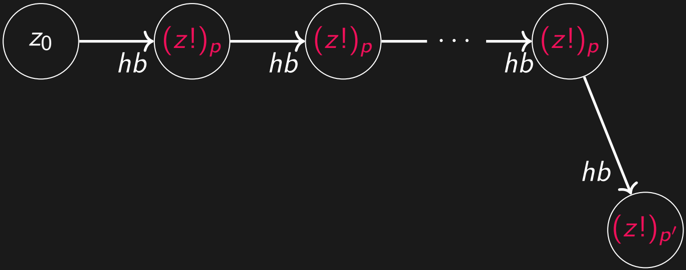
$$z_0 \xrightarrow[hb]{} (z!)_p \xrightarrow[hb]{} (z!)_p \longrightarrow \cdots \xrightarrow[hb]{} (z!)_p$$


z_0
hb
$(z!)_p$
hb
$(z!)_p$
$\cdots$
hb
$(z!)_p$
$(z!)_{p'}$


z_0
hb
$(z!)_p$
hb
$(z!)_p$
$\cdots$
hb
$(z!)_p$
hb
$(z!)_{p'}$


z_0
hb
$(z!)_p$
hb
$(z!)_p$
$\cdots$
hb
$(z!)_p$
hb
$(z!)_{p'}$


z_0
hb
$(z!)_p$
hb
$(z!)_p$
$\cdots$
hb
$(z!)_p$
hb
$(z!)_{p'}$


z_0
hb
$(z!)_p$
hb
$(z!)_p$
$\cdots$
hb
$(z!)_p$
hb
$(z!)_{p'}$

$$\llparenthesis E^z_{hb},\ z{:=}v \rrparenthesis$$

$$\llparenthesis E_{hb}^{z},\ z{:=}v \rrparenthesis$$

$$(m, !z),\ (m', ?z) \in E_{hb}^{z}$$

$$(\!|E^z_{hb},\ z{:=}v|\!)$$

$$(m, !z),\ (m', ?z) \in E^z_{hb}$$

Remember only the last write

$$(\!|E^z_{hb},\ z{:=}v|\!)$$

$$(m, !z),\ (m', ?z) \in E^z_{hb}$$

Remember only the last write

$$m(\!|E^r_{hb},\ z{:=}v|\!)$$

$$(\!|E_{hb}^{z},\ z{:=}v|\!)$$

$$(m, !z),\ (m', ?z) \in E_{hb}^{z}$$

Remember only the last write

$$m(\!|E_{hb}^{r},\ z{:=}v|\!)$$

$$E_{hb}^{z} \downarrow_{!}$$

$$(\!|E^z_{hb},\ z{:=}v|\!)$$

$$(m, !z),\ (m', ?z) \in E^z_{hb}$$

Remember only the last write

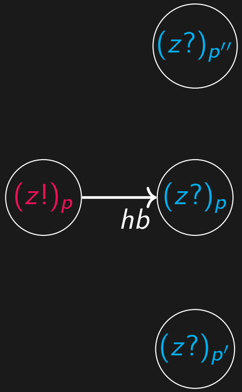
$$m(\!|E^r_{hb},\ z{:=}v|\!)$$

$$E^z_{hb}\downarrow_! \qquad (m, !z)$$

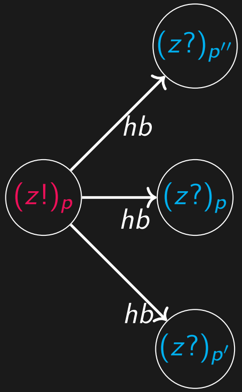
$(z!)_p$

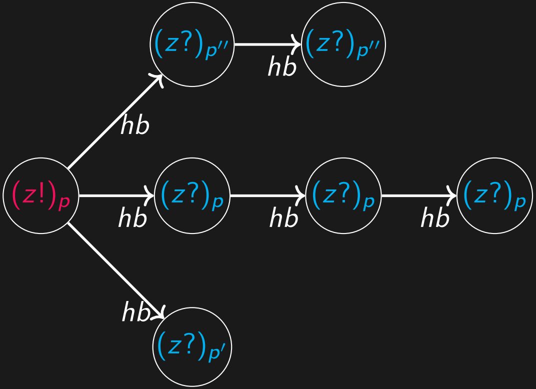
$(z!)_p$ $(z?)_p$

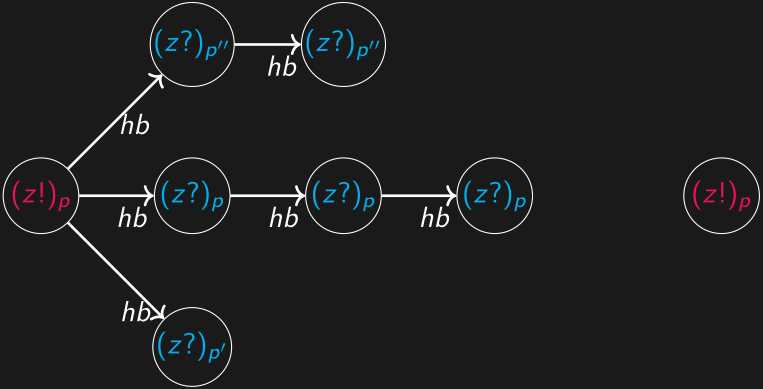
$$(z!)_p \xrightarrow[hb]{} (z?)_p$$

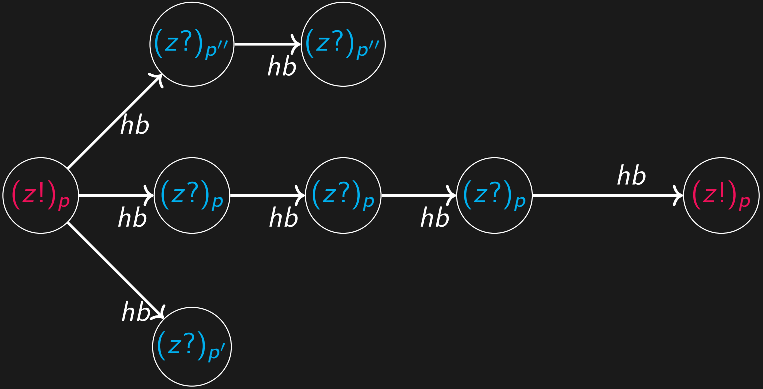


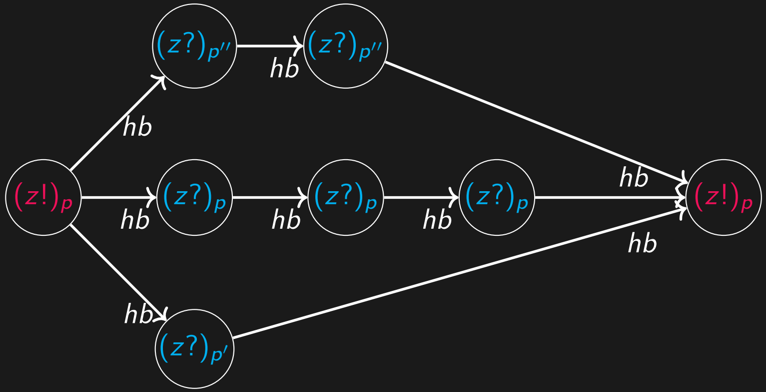
$(z?)_{p''}$
$(z!)_p$
hb
$(z?)_p$
$(z?)_{p'}$


$(z?)_{p''}$
hb
$(z!)_p$
hb
$(z?)_p$
hb
$(z?)_{p'}$

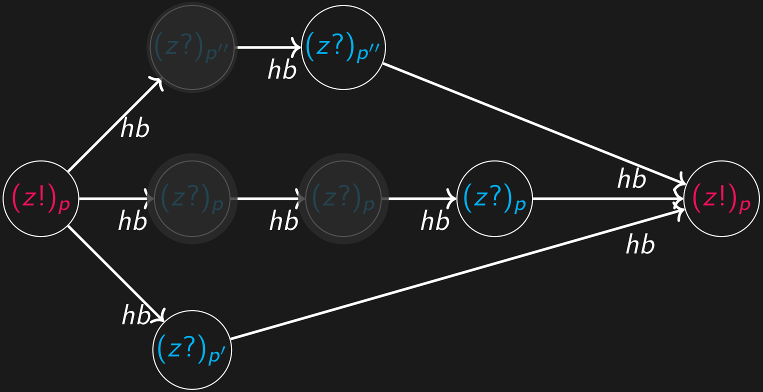

$(z!)_p$
$(z?)_{p''}$
$(z?)_{p''}$
$(z?)_p$
$(z?)_p$
$(z?)_p$
$(z?)_{p'}$
hb
hb
hb
hb
hb
hb


(z!)p
(z?)p''
(z?)p''
(z?)p
(z?)p
(z?)p
(z?)p'
(z!)p
hb
hb
hb
hb
hb
hb

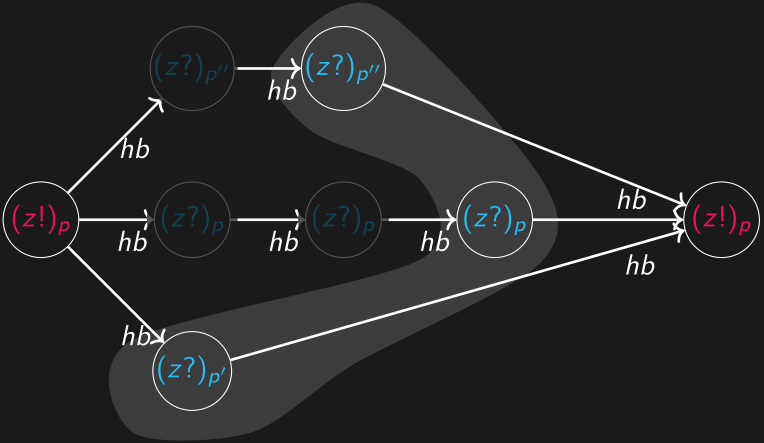

$(z?)_{p''}$
hb
$(z?)_{p''}$
hb
$(z!)_p$
hb
$(z?)_p$
hb
$(z?)_p$
hb
$(z?)_p$
hb
$(z!)_p$
hb
$(z?)_{p'}$



$(z!)_p$
$(z?)_{p''}$
$(z?)_{p''}$
$(z?)_p$
$(z?)_p$
$(z?)_p$
$(z?)_{p'}$
$(z!)_p$
hb



$(z!)_p$
$(z?)_{p''}$
$(z?)_{p''}$
$(z?)_p$
$(z?)_p$
$(z?)_p$
$(z?)_{p'}$
$(z!)_p$
hb



$(z!)_p$
$(z?)_{p''}$
$(z?)_{p''}$
$(z?)_p$
$(z?)_p$
$(z?)_p$
$(z?)_{p'}$
$(z!)_p$
hb

Updated *write*-rule for efficient race detection.

$$\frac{(m, !z) \in E_{hb} \qquad E^r_{hb} \subseteq E_{hb} \qquad \mathit{fresh}(m') \qquad E'_{hb} = \{(m', !z)\} \cup (E_{hb} - E_{hb} \downarrow_z)}{\begin{array}{ll} p\langle E_{hb}, z := v'; t\rangle & \parallel m(\!|E^r_{hb},\ z{:=}v|\!) \rightarrow \\ p\langle E'_{hb}, t\rangle & \parallel m'(\!|\emptyset,\ z{:=}v'|\!) \end{array}}$$

Updated *read*-rule for efficient race detection.

$$\frac{\begin{array}{l}(m, !z) \in E_{hb} \qquad \mathit{fresh}(m') \\ E'^{r}_{hb} = \{(m', ?z)\} \cup (E^{r}_{hb} - E_{hb}\downarrow_z) \\ E'_{hb} = \{(m', ?z)\} \cup (E_{hb} - E_{hb}\downarrow_z) \cup \{(m, !z)\}\end{array}}{\begin{array}{ll}p\langle E_{hb}, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle & \parallel m(\!|E^{r}_{hb}, z{:=}v|\!) \rightarrow \\ p\langle E'_{hb}, \mathtt{let}\ r = \ v\ \mathtt{in}\ t\rangle & \parallel m(\!|E'^{r}_{hb}, z{:=}v|\!)\end{array}}$$

Offline garbage-collection.

$$\frac{\begin{array}{rl} E'_{hb} = E_{hb} & - \{(\hat{m}, !z) \mid (\hat{m}, !z) \in E_{hb} \ \wedge \ \hat{m} \neq m\} \\ & - \{(\hat{m}, ?z) \mid (\hat{m}, ?z) \in E_{hb} \ \wedge \ (\hat{m}, ?z) \notin E^r_{hb}\} \end{array}}{p\langle E_{hb}, t\rangle \parallel m(\!|E^r_{hb},\ z{:=}v|\!) \to p\langle E'_{hb}, t\rangle \parallel m(\!|E^r_{hb},\ z{:=}v|\!)}$$

Vector clocks, Djit+ and FastTrack.

Think of a clock as natural number.
A *vector clock* maps a thread id to a clock.

Vector clocks, Djit$^+$ and FastTrack.

Think of a clock as natural number.
A *vector clock* maps a thread id to a clock.

In Djit$^+$ and FastTrack, each thread u has a vector clock C_u,

Vector clocks, Djit^{+} and FastTrack.

Think of a clock as natural number.
A *vector clock* maps a thread id to a clock.

In Djit^{+} and FastTrack, each thread u has a vector clock C_u, where it keeps:

- its own time,

Vector clocks, Djit$^+$ and FastTrack.

Think of a clock as natural number.
A *vector clock* maps a thread id to a clock.

In Djit$^+$ and FastTrack, each thread u has a vector clock C_u, where it keeps:

- its own time,
- the time of the most recent operation by v known to u.

Vector clocks, Djit$^+$ and FastTrack.

Think of a clock as natural number.
A *vector clock* maps a thread id to a clock.

In Djit$^+$ and FastTrack, each thread u has a vector clock C_u, where it keeps:

- its own time,
- the time of the most recent operation by v known to u.

$$C_u$$

Vector clocks, Djit$^+$ and FastTrack.

Think of a clock as natural number.
A *vector clock* maps a thread id to a clock.

In Djit$^+$ and FastTrack, each thread u has a vector clock C_u, where it keeps:

- its own time,
- the time of the most recent operation by v known to u.

$$C_u(v)$$

Complexity analysis

Memory per thread	Djit$^+$ FastTrack	Our approach

Complexity analysis

Memory per thread	Djit^{+} FastTrack	Our approach
worst-case	$O(\tau)$	$O(\nu\tau)$

τ number of threads

ν number of variables

Complexity analysis

Memory per thread	Djit$^+$ FastTrack	Our approach
worst-case	$O(\tau)$	$O(\nu\tau)$
best-case	$O(1)$	

τ number of threads

ν number of variables

Complexity analysis

Memory per thread	Djit$^+$ FastTrack	Our approach
worst-case	$O(\tau)$	$O(\nu\tau)$
best-case	$O(1)$	$O(1)$

τ number of threads

ν number of variables

Comparison with TSan.

Also in the paper

Rules for

- synchronous communication
- dynamic channel and thread creation
- data-race reporting, etc

Connection with trace theory

Summary

Summary

Data-race detection in terms of channel communication

Summary

Data-race detection in terms of channel communication

- message passing as synchronization primitive

Summary

Data-race detection in terms of channel communication

- message passing as synchronization primitive
- no vector-clocks; based directly on happens-before relation

Summary

Data-race detection in terms of channel communication

- message passing as synchronization primitive
- no vector-clocks; based directly on happens-before relation
- most recent write, most recent reads, garbage collection

Summary

Data-race detection in terms of channel communication

- message passing as synchronization primitive
- no vector-clocks; based directly on happens-before relation
- most recent write, most recent reads, garbage collection
- models happens-before as described by the Go memory model

Future work

Future work

- Implementation

Future work

- Implementation
- Proof of "minimality of information"

Future work

- Implementation
- Proof of "minimality of information"

  Least amount of event information that must be kept

  Largest amount of information that can be garbage collected

Future work

- Implementation
- Proof of “minimality of information”

  Least amount of event information that must be kept

  Largest amount of information that can be garbage collected

Future work

- Implementation
- Proof of "minimality of information"

  Least amount of event information that must be kept
  Largest amount of information that can be garbage collected

Questions?

References

- Go memory model (2014). The Go memory model. `https://golang.org/ref/mem`. Version of May 31, 2014, covering Go version 1.9.1
- Fava, D. and Steffen, M. (2019). Ready, set, Go! Data-race detection and the Go language. *To appear in the pre-proceedings of the Brazilian Symposium on Formal Methods (SBMF)*. `http://arxiv.org/abs/1910.12643`

References

- Fava, D., Steffen, M., and Stolz, V. (2018b). Operational semantics of a weak memory model with channel synchronization.
  Journal of Logic and Algebraic Methods in Programming.
  An extended version of the FM'18 publication with the same title

- Fava, D., Steffen, M., and Stolz, V. (2018a). Operational semantics of a weak memory model with channel synchronization.
  In Havelund, K., Peleska, J., Roscoe, B., and de Vink, E., editors, *FM*, volume 10951 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag

- French, R. Gopher figure by Renee French. https://blog.golang.org/gopher