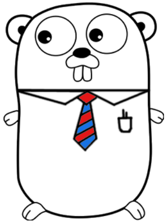
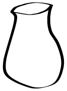
Operational Semantics of a Weak Memory Model with Channel Communication

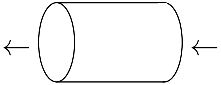
Daniel S. Fava
Martin Steffen
Volker Stolz

Department of informatics
University of Oslo, Norway





memory
model



channel
communication

$$\frac{\bullet \Rightarrow \blacksquare \quad \bullet}{\blacksquare}$$

operational
semantics

What’s a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

What's a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

- In a single thread case $\Rightarrow$ program order

What's a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

- In a single thread case ⇒ program order

```
        T0
z    := 42
flag := 1
load z
```

What's a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

- In a single thread case ⇒ program order

```
         T0
z    := 42
flag := 1
load z
```

z =?

What’s a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

- In a single thread case $\Rightarrow$ program order

```
          T0
z    := 42
flag := 1
load z
```

$z = 42$

What’s a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

- In a single thread case ⇒ program order
- Informs how threads interact through shared memory

```
     T0
z    := 42
flag := 1
load z
```

$z = 42$

What's a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

- In a single thread case ⇒ program order
- Informs how threads interact through shared memory

```
     T0           |      T1
z    := 42        |
flag := 1         |   load flag
load z            |   load z
```

$z = 42$

What's a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

- In a single thread case $\Rightarrow$ program order
- Informs how threads interact through shared memory

Initially $\mathtt{z} = 0$

```
      T0               |       T1
z    := 42             |
flag := 1              |    load flag
load z                 |    load z
```

$\mathtt{z} = 42$

What's a memory model?

A memory model dictates what values can be read from memory at a given point in the execution

- In a single thread case $\Rightarrow$ program order
- Informs how threads interact through shared memory

Initially $\texttt{z} = 0$

```
      T0                 |        T1
z    := 42               |
flag := 1                |     load flag
load z                   |     load z
```

$\texttt{z} = 42$ $\qquad$ $\texttt{flag} = 1 \;\Rightarrow\; \texttt{z} = ?$

Sequential consistency

Memory as a shared global repository where operations appear atomic and in program order

Sequential consistency

Memory as a shared global repository where operations appear atomic and in program order

+ Simple to reason bout

Initially $z = 0$

```
      T0            |        T1
z    := 42          |
flag := 1           |     load flag
load z              |     load z
```

$z = 42$ $\qquad\qquad$ $flag = 1 \Rightarrow z = 42$

Sequential consistency

Memory as a shared global repository where operations appear atomic and in program order

+ Simple to reason bout
- Does not reflect modern hardware
- Restricts compiler optimizations

Initially $\mathtt{z} = 0$

```
     T0               |        T1
z    := 42            |
flag := 1             |     load flag
load z                |     load z
```

$\mathtt{z} = 42$ $\qquad\qquad$ $\mathtt{flag} = 1 \;\Rightarrow\; \mathtt{z} = 42$

Weak memory

Weak memory

- Relaxations to the order of memory operations

Weak memory

- Relaxations to the order of memory operations
- Motivated by efficiency (synchronize only when needed)

Weak memory

- Relaxations to the order of memory operations
- Motivated by efficiency (synchronize only when needed)
- Cognitive burden placed on the programmer

Weak memory

- Relaxations to the order of memory operations
- Motivated by efficiency (synchronize only when needed)
- Cognitive burden placed on the programmer

Initially $\texttt{z} = 0$

```
     T0            |        T1
z    := 42         |
flag := 1          |    load flag
load z             |    load z
```

$\texttt{z} = 42$ $\texttt{flag} = 1 \;\Rightarrow\; \texttt{z} \in \{0, 42\}$

- Memory models often focus on locks, barriers, semaphores as synchronization primitives
- Their formalization is often axiomatic

- Memory models often focus on locks, barriers, semaphores as synchronization primitives
- Their formalization is often axiomatic

Our motivation is to formalize a weak memory model

- Memory models often focus on locks, barriers, semaphores as synchronization primitives
- Their formalization is often axiomatic

Our motivation is to formalize a weak memory model

operationally

- Memory models often focus on locks, barriers, semaphores as synchronization primitives
- Their formalization is often axiomatic

Our motivation is to formalize a weak memory model

operationally (intuitive)

- Memory models often focus on locks, barriers, semaphores as synchronization primitives
- Their formalization is often axiomatic

Our motivation is to formalize a weak memory model

operationally, and focusing on (intuitive)

channel communication for synchronization

- Memory models often focus on locks, barriers, semaphores as synchronization primitives
- Their formalization is often axiomatic

Our motivation is to formalize a weak memory model

operationally, and focusing on (intuitive)

channel communication for synchronization (novel)

- Memory models often focus on locks, barriers, semaphores as synchronization primitives
- Their formalization is often axiomatic

Our motivation is to formalize a weak memory model

operationally, and focusing on (intuitive)

channel communication for synchronization (novel)

We took inspiration from the Go language

Models often described from a hardware-centric perspective

- Write buffers, caches, (pipeline) flushes, etc.

Models often described from a hardware-centric perspective

- Write buffers, caches, (pipeline) flushes, etc.

We took a “software” perspective

Models often described from a hardware-centric perspective

- Write buffers, caches, (pipeline) flushes, etc.

We took a “software” perspective

- focus on reasoning about program behavior

Models often described from a hardware-centric perspective

- Write buffers, caches, (pipeline) flushes, etc.

We took a “software” perspective

- focus on reasoning about program behavior
- account for hardware and compiler implementations

Models often described from a hardware-centric perspective

- Write buffers, caches, (pipeline) flushes, etc.

We took a "software" perspective

- focus on reasoning about program behavior
- account for hardware and compiler implementations
- but not concerned with being "implementable"

Models often described from a hardware-centric perspective

- Write buffers, caches, (pipeline) flushes, etc.

We took a “software” perspective

- focus on reasoning about program behavior
- account for hardware and compiler implementations
- but not concerned with being “implementable”

  Freed us to think about a (potentially) simpler model

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

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

- *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.*

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

- *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.*

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

Order and Observability

- *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.*

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

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\mathsf{hb}} e'$

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\mathsf{hb}} e'$

```
      T0
z    := 42
flag := 1
```

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\mathsf{hb}} e'$

```
        T0
z    := 42   (A)
flag := 1    (B)
```

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\text{hb}} e'$

```
        T0
z     := 42   (A)
flag  := 1    (B)
```

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

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\text{hb}} e'$

```
          T0               |          T1
z    := 42   (A)           |    load flag
flag := 1    (B)           |    load z
```

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

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\mathsf{hb}} e'$

```
          T0                 |          T1
z     := 42   (A)            |    load flag   (C)
flag  := 1    (B)            |    load z      (D)
```

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

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\mathsf{hb}} e'$

```
       T0              |         T1
z    := 42   (A)       |    load flag   (C)
flag := 1    (B)       |    load z      (D)
```

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

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\mathrm{hb}} e'$

```
          T0              |          T1
z    := 42   (A)          |   load flag   (C)
flag := 1    (B)          |   load z      (D)
```

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

-
-

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\mathsf{hb}} e'$

```
      T0             |        T1
z    := 42   (A)     |   load flag   (C)
flag := 1    (B)     |   load z      (D)
```

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

- Just because $\mathtt{A} \rightarrow_{\mathsf{hb}} \mathtt{B}$, it does not mean A occurred before B
-

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\text{hb}} e'$

```
      T0               |         T1
z    := 42   (A)       |   load flag   (C)
flag := 1    (B)       |   load z      (D)
```

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

- Just because $\texttt{A} \rightarrow_{\text{hb}} \texttt{B}$, it does not mean `A` occurred before `B`
- Just because `B` occurred before `C`, it does not mean $\texttt{B} \rightarrow_{\text{hb}} \texttt{C}$

Order

Happens-before relation [Lamport, 1978]

A relation on events. $e \rightarrow_{\mathsf{hb}} e'$

```
       T0                |       T1
z     := 42   (A)        |   load flag   (C)
flag  := 1    (B)        |   load z      (D)
```

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

- Just because $\mathtt{A} \rightarrow_{\mathsf{hb}} \mathtt{B}$, it does not mean `A` occurred before `B`
- Just because `B` occurred before `C`, it does not mean $\mathtt{B} \rightarrow_{\mathsf{hb}} \mathtt{C}$

No ordering between events of different threads

$$\mathtt{A} \rightarrow_{\mathsf{hb}} \mathtt{B} \wedge \mathtt{C} \rightarrow_{\mathsf{hb}} \mathtt{D} \quad \not\Rightarrow \quad \mathtt{A} \rightarrow_{\mathsf{hb}} \mathtt{D}$$

Observability

Observability

Observability is defined negatively

Observability

Observability is defined negatively

A read r of variable z can observe a write w also to z **unless**:

- $r \rightarrow_{\text{hb}} w$
- $w \rightarrow_{\text{hb}} w' \rightarrow_{\text{hb}} r$
  for some write w' to z

[Go memory model, 2014]

Observability

Observability is defined negatively

A read r of variable z can observe a write w also to z **unless**:

- $r \rightarrow_{\text{hb}} w$
- $w \rightarrow_{\text{hb}} w' \rightarrow_{\text{hb}} r$ for some write w' to z

[Go memory model, 2014]

```
   T
load z
z := 42
```

Observability

Observability is defined negatively

A read r of variable z can observe a write w also to z **unless**:

- $r \rightarrow_{\text{hb}} w$
- $w \rightarrow_{\text{hb}} w' \rightarrow_{\text{hb}} r$
  for some write w' to z

[Go memory model, 2014]

```
    T
load z    (A)
z := 42   (B)
```

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

Observability

Observability is defined negatively

A read r of variable z can observe a write w also to z **unless**:

- $r \rightarrow_{\text{hb}} w$
- $w \rightarrow_{\text{hb}} w' \rightarrow_{\text{hb}} r$
  for some write w' to z

[Go memory model, 2014]

```
   T
load z    (A)
z := 42   (B)
```

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

```
  T'
z := 1
z := 2
load z
```

Observability

Observability is defined negatively

A read r of variable z can observe a write w also to z **unless**:

- $r \rightarrow_{\text{hb}} w$
- $w \rightarrow_{\text{hb}} w' \rightarrow_{\text{hb}} r$ for some write w' to z

[Go memory model, 2014]

```
   T
load z    (A)
z := 42   (B)
```

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

```
   T'
z := 1   (A')
z := 2   (B')
load z   (C')
```

$\mathtt{A'} \rightarrow_{\text{hb}} \mathtt{B'} \rightarrow_{\text{hb}} \mathtt{C'}$

Observability

Observability is defined negatively

A read r of variable z can observe a write w also to z **unless**:

- $r \rightarrow_{\mathrm{hb}} w$
- $w \rightarrow_{\mathrm{hb}} w' \rightarrow_{\mathrm{hb}} r$ for some write w' to z

[Go memory model, 2014]

```
   T
load z    (A)
z := 42   (B)
```

`A` $\rightarrow_{\mathrm{hb}}$ `B`

```
   T'
z := 1   (A')
z := 2   (B')
load z   (C')
```

`A'` $\rightarrow_{\mathrm{hb}}$ `B'` $\rightarrow_{\mathrm{hb}}$ `C'`

`B'` shadows `A'`

Observability

Observability is defined negatively

A read r of variable z can observe a write w also to z **unless**:

- $r \rightarrow_{\text{hb}} w$
- $w \rightarrow_{\text{hb}} w' \rightarrow_{\text{hb}} r$ for some write w' to z

[Go memory model, 2014]

```
   T
load z    (A)
z := 42   (B)
```

A $\rightarrow_{\text{hb}}$ B

```
   T'
z := 1   (A')
z := 2   (B')
load z   (C')
```

A' $\rightarrow_{\text{hb}}$ B' $\rightarrow_{\text{hb}}$ C'

B' shadows A'

relative to a thread

Our model

Our model

Memory is a set of write events $m(z{:=}v)$

Our model

Memory is a set of write events $m(z{:=}v)$

Each thread keeps track of:

events in its past	(happened-before set)
un-observable events	(shadowed set)

Our model

Memory is a set of write events $m(z{:=}v)$

Each thread keeps track of:

events in its past (happened-before set)

un-observable events (shadowed set)

When a thread reads from memory:

Our model

Memory is a set of write events $m(z{:=}v)$

Each thread keeps track of:

events in its past (happened-before set)

un-observable events (shadowed set)

When a thread reads from memory:

read any write event that is not in its *shadowed set*

Our model

Memory is a set of write events $m(z{:=}v)$

Each thread keeps track of:

events in its past (happened-before set)

un-observable events (shadowed set)

When a thread reads from memory:

read any write event that is not in its *shadowed set*

When a thread writes to memory it update its local state:

Our model

Memory is a set of write events $m(z:=v)$

Each thread keeps track of:

events in its past (happened-before set)

un-observable events (shadowed set)

When a thread reads from memory:

read any write event that is not in its *shadowed set*

When a thread writes to memory it update its local state:

recording the write as having happened in the past

Our model

Memory is a set of write events $m(z{:=}v)$

Each thread keeps track of:

- events in its past (happened-before set)
- un-observable events (shadowed set)

When a thread reads from memory:

- read any write event that is not in its *shadowed set*

When a thread writes to memory it update its local state:

- recording the write as having happened in the past
- recording writes that became un-observable

$$\frac{}{p\langle \sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle}\ \textsc{Read}$$

$$\frac{}{p\langle \sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel m(\!|z{:=}v|\!)}\ \textsc{Read}$$

$$\frac{}{\begin{array}{c} p\langle\sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel m(\!|z{:=}v|\!) \\ \Rightarrow \quad p\langle\sigma, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel m(\!|z{:=}v|\!) \end{array}}\ \textsc{Read}$$

$$\frac{\sigma = (\_, E_s) \qquad m \notin E_s}{\begin{array}{c} p\langle \sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel m(\!|z{:=}v|\!) \\ \Rightarrow \quad p\langle \sigma, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel m(\!|z{:=}v|\!) \end{array}}\ \textsc{Read}$$

$$\frac{}{p\langle \sigma, z := v; t\rangle}\ \textsc{Write}$$

$$\frac{}{p\langle \sigma, z := v; t\rangle \quad \Rightarrow \quad p\langle \sigma', t\rangle \parallel m(\!|z{:=}v|\!)}\ \textsc{Write}$$

$$\frac{\mathit{fresh}(m)}{p\langle\sigma, z := v; t\rangle \quad\Rightarrow\quad p\langle\sigma', t\rangle \parallel m\llparenthesis z{:=}v\rrparenthesis}\ \textsc{Write}$$

$$\dfrac{\sigma = (E_{hb}, E_s) \qquad\qquad \mathit{fresh}(m)}{p\langle \sigma, z := v; t\rangle \quad \Rightarrow \quad p\langle \sigma', t\rangle \parallel m(\!|z{:=}v|\!)}\ \textsc{Write}$$

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (m, z), E_s + E_{hb}(z)) \qquad \mathit{fresh}(m)}{p\langle \sigma, z := v; t\rangle \quad \Rightarrow \quad p\langle \sigma', t\rangle \parallel m(\!| z{:=}v |\!)}\ \textsc{Write}$$

Synchronization

Motivating example

```
  Producer             |      Consumer
z    := 42   (A)       |  while (flag != 1) {}     (C)
flag := 1    (B)       |  load z                   (D)
```

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

```
  Producer              |      Consumer
z    := 42   (A)        |  while (flag != 1) {}     (C)
flag := 1    (B)        |  load z                   (D)
```

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

$B \underset{?}{\rightarrow_{hb}} C$

```
  Producer              |      Consumer
z    := 42   (A)        |  while (flag != 1) {}     (C)
flag := 1    (B)        |  load z                   (D)
```

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

$\mathtt{B} \underset{?}{\rightarrow_{\mathsf{hb}}} \mathtt{C}$

Synchronization via channel communication

```
  Producer             |      Consumer
z    := 42  (A)        |  while (flag != 1) {}    (C)
            (B)        |  load z                  (D)
```

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

$\mathtt{B} \underset{?}{\rightarrow_{\mathsf{hb}}} \mathtt{C}$

Synchronization via channel communication

```
  Producer             |      Consumer
z    := 42   (A)       |  while (flag != 1) {}     (C)
c    <- 0    (B)       |  load z                   (D)
```

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

$B \underset{?}{\rightarrow_{\mathsf{hb}}} C$

Synchronization via channel communication

```
  Producer            |     Consumer
z    := 42  (A)       |                                (C)
c    <- 0   (B)       |  load z                        (D)
```

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

$\mathtt{B} \underset{?}{\rightarrow_{\mathsf{hb}}} \mathtt{C}$

Synchronization via channel communication

```
  Producer            |     Consumer
z    := 42   (A)      |  <- c                     (C)
c    <- 0    (B)      |  load z                   (D)
```

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

$\mathtt{B} \underset{\mathbf{?}}{\rightarrow_{\mathsf{hb}}} \mathtt{C}$

Synchronization via channel communication

```
  Producer              |      Consumer
z    := 42  (A)         |  <- c                    (C)
c    <- 0   (B)         |  load z                  (D)
```

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

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

Synchronization via channel communication

```
  Producer              |      Consumer
z    := 42  (A)         |  <- c                       (C)
c    <- 0   (B)         |  load z                     (D)
```

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

$B \rightarrow_{\mathsf{hb}} C$

Synchronization via channel communication

$A \rightarrow_{\mathsf{hb}} D$

$$\frac{}{p\langle \sigma, c \leftarrow v; t\rangle \quad \| \; c[q]}\;\textsc{Send}$$

$$\frac{}{\begin{array}{ll} p\langle \sigma, c \leftarrow v; t\rangle & \parallel c[q] \quad \Rightarrow \\ p\langle \sigma', t\rangle & \parallel c[(v, \sigma) :: q] \end{array}} \textsc{Send}$$

$$\frac{\neg closed(c[q_2]) \qquad \sigma' = \sigma + \sigma''}{\begin{array}{rcl} c_b[q_1 :: \sigma''] \parallel & p\langle \sigma, c \leftarrow v; t\rangle & \parallel c[q_2] \quad \Rightarrow \\ c_b[q_1] \parallel & p\langle \sigma', t\rangle & \parallel c[(v, \sigma) :: q_2] \end{array}} \textsc{Send}$$

$$\frac{}{p\langle \sigma, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle \quad \| \ c[q :: (v, \sigma'')]}\ \textsc{Rec}$$

$$\frac{\sigma' = \sigma + \sigma''}{\begin{array}{lll} p\langle \sigma, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle & \parallel c[q :: (v, \sigma'')] & \Rightarrow \\ p\langle \sigma', \texttt{let}\ r = v\ \texttt{in}\ t\rangle & \parallel c[q] & \end{array}}\ \textsc{Rec}$$

$$\frac{v \neq \bot \qquad \sigma' = \sigma + \sigma''}{\begin{array}{r}c_b[q_1] \parallel \quad p\langle \sigma, \texttt{let } r = \leftarrow c \texttt{ in } t\rangle \quad \parallel c[q_2 :: (v, \sigma'')] \quad \Rightarrow \\ c_b[\sigma :: q_1] \parallel \quad p\langle \sigma', \texttt{let } r = v \texttt{ in } t\rangle \quad \parallel c[q_2] \qquad\qquad\end{array}}\ \textsc{Rec}$$

We have seen so far,

We have seen so far,

Operational semantics of a weak memory model with channels

We have seen so far,

Operational semantics of a weak memory model with channels

- Memory is a set of write events
  as opposed to a mapping of variables to values

We have seen so far,

Operational semantics of a weak memory model with channels

- Memory is a set of write events
  as opposed to a mapping of variables to values
- Writes become observable *globally* and *immediately*

We have seen so far,

Operational semantics of a weak memory model with channels

- Memory is a set of write events
  as opposed to a mapping of variables to values
- Writes become observable *globally* and *immediately*
- Order (HB) and observability (shadowing)

We have seen so far,

Operational semantics of a weak memory model with channels

- Memory is a set of write events
  as opposed to a mapping of variables to values
- Writes become observable *globally* and *immediately*
- Order (HB) and observability (shadowing)
    - is *thread local*

We have seen so far,

Operational semantics of a weak memory model with channels

- Memory is a set of write events
  as opposed to a mapping of variables to values
- Writes become observable *globally* and *immediately*
- Order (HB) and observability (shadowing)
    - is *thread local*
    - and travels on channels, implicitly

We have seen so far,

Operational semantics of a weak memory model with channels

- Memory is a set of write events
  as opposed to a mapping of variables to values
- Writes become observable *globally* and *immediately*
- Order (HB) and observability (shadowing)
  - is *thread local*
  - and travels on channels, implicitly

Synchronization as restriction on observability

Correctness

Relating the weak model to a sequentially consistent model

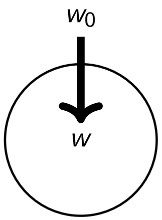
P

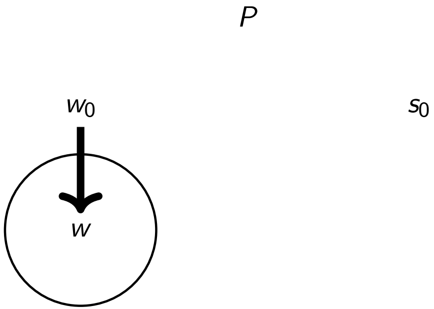
P

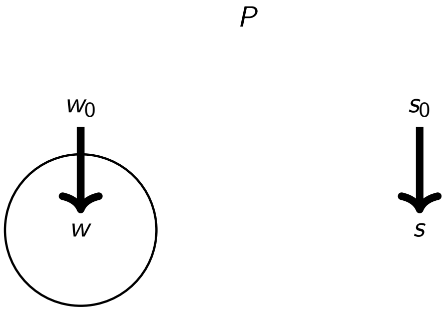
w_0

$$\begin{tikzcd} & P \\ w_0 \arrow[d] & \\ w & \end{tikzcd}$$

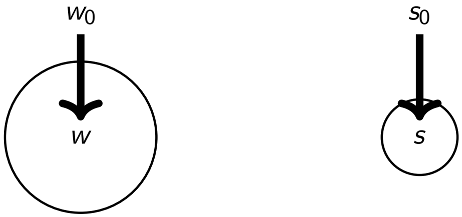
P



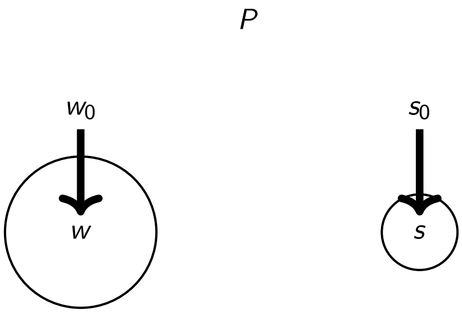

P
w_0
s_0
w


P
w_0
w
s_0
s

P

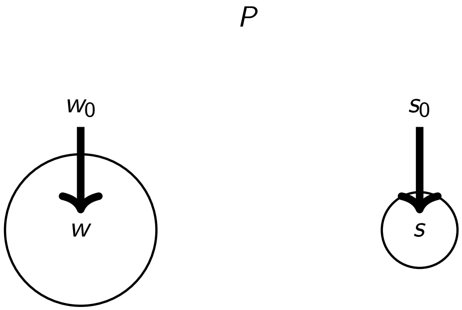


A property desired of weak memory models



A property desired of weak memory models

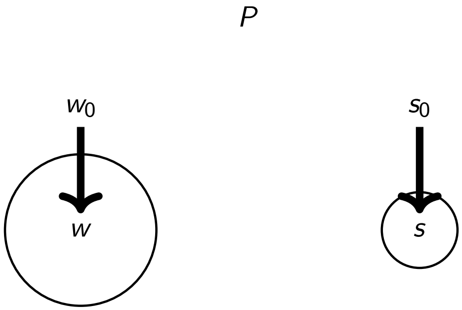
Sequentially consistent data-race free (SC-DRF) guarantee



A property desired of weak memory models

Sequentially consistent data-race free (SC-DRF) guarantee

- Allows programmers to think in terms of strong memory
- Write it once, run it everywhere
    - provided program is DRF and memory models are SC-DRF

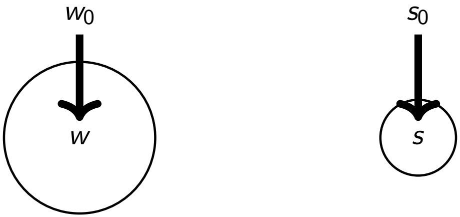


A property desired of weak memory models

Sequentially consistent data-race free (SC-DRF) guarantee

- Allows programmers to think in terms of strong memory
- Write it once, run it everywhere
    - provided program is DRF and memory models are SC-DRF

$DRF(P)$



A property desired of weak memory models

Sequentially consistent data-race free (SC-DRF) guarantee

- Allows programmers to think in terms of strong memory
- Write it once, run it everywhere
  - provided program is DRF and memory models are SC-DRF

$DRF(P)$

We prove a desired property of the model

Sequentially consistent data-race free (SC-DRF) guarantee

We prove a desired property of the model

Sequentially consistent data-race free (SC-DRF) guarantee

Proof of conditional simulation

We prove a desired property of the model

Sequentially consistent data-race free (SC-DRF) guarantee

Proof of conditional simulation

$$DRF(P)$$

We prove a desired property of the model

Sequentially consistent data-race free (SC-DRF) guarantee

Proof of conditional simulation

$$DRF(P)$$

$$\begin{tikzcd} w_0 \arrow[r, dash] & \mathcal{R} \arrow[r, dash] & s_0 \end{tikzcd}$$

We prove a desired property of the model

Sequentially consistent data-race free (SC-DRF) guarantee

Proof of conditional simulation

$$DRF(P)$$

$$\begin{tikzcd} w_0 \arrow[r, dash] \arrow[d, "\alpha"'] & \mathcal{R} \arrow[r, dash] & s_0 \\ w & & \end{tikzcd}$$

We prove a desired property of the model

Sequentially consistent data-race free (SC-DRF) guarantee

Proof of conditional simulation

$DRF(P)$

$$\begin{tikzcd}
w_0 \arrow[r, dash, "\mathcal{R}" description] \arrow[d, "\alpha"'] & s_0 \arrow[d, dotted, "\alpha"'] \\
w & s
\end{tikzcd}$$

We prove a desired property of the model

Sequentially consistent data-race free (SC-DRF) guarantee

Proof of conditional simulation

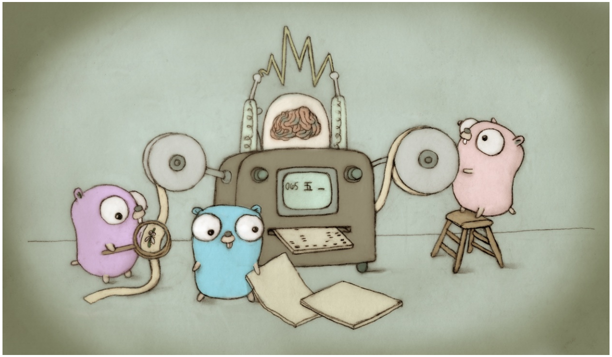
$DRF(P)$

$$\begin{tikzcd}
w_0 \arrow[rr, dash, "\mathcal{R}" description] \arrow[d, "\alpha"'] & & s_0 \arrow[d, dotted, "\alpha"'] \\
w \arrow[rr, dash, dotted, "\mathcal{R}" description] & & s
\end{tikzcd}$$

We have implemented our semantics in $\mathbb{K}$, which is an executable semantics framework

Available on the mmGo GitHub page
`https://github.com/dfava/mmgo`

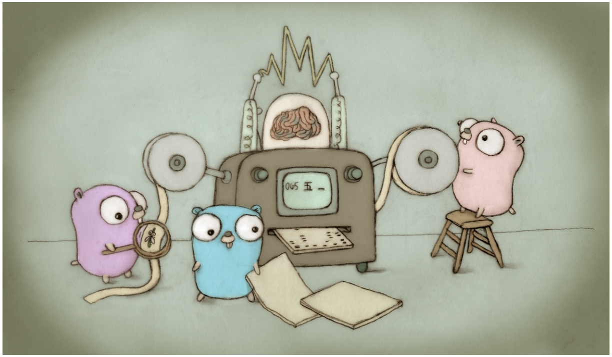
Hypothesis. We can use the semantics for data-race detection



Hypothesis. We can use the semantics for data-race detection

Current goal. To relax model by accounting for read buffers

i.e. branching on values read but not yet "resolved"



In summary,

In summary,

- week memory model

In summary,

- week memory model
- operational semantics

In summary,

- week memory model
- operational semantics
- channel communication as synchronization primitive

In summary,

- week memory model
- operational semantics
- channel communication as synchronization primitive
- proof of SC-DRF guarantee

In summary,

- week memory model
- operational semantics
- channel communication as synchronization primitive
- proof of SC-DRF guarantee
- pointer to an implementation

In summary,

- week memory model
- operational semantics
- channel communication as synchronization primitive
- proof of SC-DRF guarantee
- pointer to an implementation

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
- Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system.
  Communications of the ACM, 21(7):558–565
- French, R. Gopher figure by Renee French.
  https://blog.golang.org/gopher