Operational Semantics of a Weak Memory Model with Channel Communication

Daniel Fava, Martin Steffen, Volker Stolz

1st March 2018

In this talk

- *operational* semantics for a WMM
  - inspired by Go
  - channel communication
  - based on happens-before
- proof of basic *correctness* property
- executable within the $\mathbb{K}$ rewriting framework

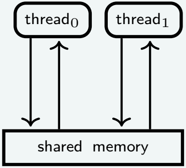
Memory model

MCM

A specification what to expect from from a shared mamory, what may be observed (by reads) and what not.

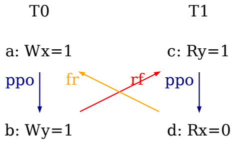
Rest

- bottom-line: *sequential consistency* Lamport (interleaving of reads and writes),
- *weak* or *relaxed*: basically weaker than that.

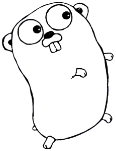


How to specify a MM

- in prose (as in `https://golang.org/ref/mem`)
- litmus tests
- axiomatic (candidate executions)
- operational (SOS)



Go sales pitch



- “language for the 21st century”
- relatively new language (with some not so new features?)
- a lot of *fanfare* & backed by Google no less
- existing show-case applications
  - docker
  - dropbox . . .

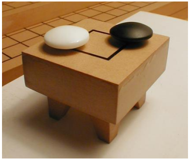
Go's stated design principles

- appealing to C programmers
- KISS: “keep it simple, stupid”
- built-in concurrency
- “strongly typed”
- efficient
- fast *compilation*, appealing for scripting

Go's non-revolutionary feature mix

- imperative
- object-oriented (?)
- compiled
- concurrent (goroutines)
- "strongishly" typed
- garbage collected
- portable
- higher-order functions and closures

Calculus



- simple concurrent calculus with “goroutines”
- A-normal form
- channels:
  - dynamically created
  - “higher-order” channels (à la π ...)
  - *bounded* channels
  - mixed choice
    - with “channel guards”
    - default-clause

Syntax

$$
\begin{array}{llll}
v & ::= & r \mid \underline{n} & \text{values} \\
e & ::= & t \mid v \mid \texttt{load}\ z \mid z := v \mid \texttt{if}\ v\ \texttt{then}\ t\ \texttt{else}\ t \mid \text{go}\ t & \text{expression} \\
 & \mid & \texttt{make}\ (\texttt{chan}\ T, v) \mid \leftarrow v \mid v \leftarrow v \mid \texttt{close}\ v & \\
g & ::= & v \leftarrow v \mid \leftarrow v \mid \texttt{default} & \text{guards} \\
t & ::= & \texttt{let}\ r = e\ \texttt{in}\ t \mid \sum_i \texttt{let}\ r_i = g_i\ \texttt{in}\ t_i & \text{threads}
\end{array}
$$

- $\sum$: choice (`select`, `case`, `default`)

Go's concurrency model

- only sync-primitive: *channel* communication (but read the fine-print)
- shared variable communication possible ⇒
- simple happens-before memory model

Mantra

Don't communicate by sharing memory; share memory by communicating. (R. Pike)

Rest

- straighforward and simple model, still they advise:

  "If you must read the rest of this document [= the Go MM] to understand the behavior of your program, you are being too clever. Don't be clever."

Happens-before

- dates back to Lamport
- "unrelated" to actually "happening before"

Observational + "liberal"

a *read* can observe a write W unless

1. read definitely *"too late"*
2. a different write definitely "overwrites" W (shadows)

Nature of synchronization

- generally:[1] restricting otherwise possible *interleavings*
- in connection with shared memory: intuition often (cf. *write buffers*)

  "Data Memory Barrier (DMB). This forces all earlier-in-program-order memory accesses to become globally visible before any subsequent accesses." (random quote, some ARM programmer's guide)

[1] independent from shared memory

Nature of synchronization

- generally:[1] restricting otherwise possible *interleavings*
- in connection with shared memory: intuition often (cf. *write buffers*)

  "Data Memory Barrier (DMB). This forces all earlier-in-program-order memory accesses to become globally visible before any subsequent accesses." (random quote, some ARM programmer's guide)

Happens-before

synchronization = making things INVISBLE

[1] independent from shared memory

Two(*) ingredients for HB only

1. *program* order
2. channel communication
   2.1 sending $\rightarrow_{\mathsf{hb}}$ recieving
   2.2 full buffer

Sends and receives

- A *send* on a channel happens-before the corresponding *receive* from that channel completes.
- The ith *receive* on a channel with capacity k happens-before the $i+k$th *send* from that channel completes.

Rest

- channel close, init, thread creation, packages, locks, once

Operational semantics (weak)

Configuration

$$P ::= n\langle \sigma, t\rangle \mid n(\!|z{:=}v|\!) \mid \bullet \mid P \parallel P \mid n[q] \mid \nu n\, P\,. \quad (1)$$

Operational semantics (weak)

Configuration

$$P ::= n\langle \sigma, t\rangle \mid n(\!| z{:=}v |\!) \mid \bullet \mid P \parallel P \mid n[q] \mid \nu n\, P\,. \quad (1)$$

thread

$$n\langle \sigma, t\rangle$$

Operational semantics (weak)

Configuration

$$P ::= n\langle\sigma, t\rangle \mid n(\!|z{:=}v|\!) \mid \bullet \mid P \parallel P \mid n[q] \mid \nu n\, P\,. \quad (1)$$

thread

$$n\langle\sigma, t\rangle$$

channel

$$n[q]$$

Operational semantics (weak)

Configuration

$$P ::= n\langle \sigma, t\rangle \mid n(\!|z{:=}v|\!) \mid \bullet \mid P \parallel P \mid n[q] \mid \nu n\, P\,. \quad (1)$$

thread

$$n\langle \sigma, t\rangle$$

write event

$$n(\!|z{:=}v|\!)$$

channel

$$n[q]$$

Write & read steps

$$\frac{\dots}{p\langle\sigma, z := v; t\rangle \quad \to \quad p\langle\sigma', t\rangle \parallel n(\!|z{:=}v|\!)}$$

$$\frac{\dots}{p\langle\sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel n(\!|z{:=}v|\!) \quad \to \quad p\langle\sigma, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel n(\!|z{:=}v|\!)}$$

Synchronization = making things unobservable

- reads and writes: no synchronization
- program order
  - the only component of happens-before
  - for `x:=1; x:=2`: value 1 unobservable

but only locally

- channel communication; only (interesting) means of synchronization

Synchronization = making things unobservable

- reads and writes: no synchronization
- program order
  - the only component of happens-before
  - for `x:=1; x:=2`: value 1 unobservable

but only locally

- channel communication; only (interesting) means of synchronization

Channel communication

- send the communicated value from sender two receiver

Synchronization = making things unobservable

- reads and writes: no synchronization
- program order
  - the only component of happens-before
  - for `x:=1; x:=2`: value 1 unobservable

but only locally

- channel communication; only (interesting) means of synchronization

Channel communication

- send the communicated value from sender two receiver
- inform receiver of local knowledge of UNobservable write events

Shadow sets and local information

- every write even: unique identifier

thread local information

1. which events are locally known to be unobservable (shadowed)
2. which events are locally known to have happened-before (at the current point)

Rest

$$n\langle \sigma, t\rangle$$

- local “state” tuple (E_{hb}, E_s), $\sigma : 2^{(N \times X)} \times 2^N$.

Read & write once more

- of course: shadow sets used to make writes invisible
- local update of "program order"

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (\mathbf{n}, \mathbf{z}), E_s + \mathbf{E}_{hb}(\mathbf{z}))}{p\langle \sigma, z := v; t\rangle \quad \rightarrow \quad p\langle \sigma', t\rangle \parallel n(\!|z{:=}v|\!)}$$

$$\frac{\sigma = (\_, E_s) \qquad \mathbf{n} \notin \mathbf{E}_s}{p\langle \sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel n(\!|z{:=}v|\!) \quad \rightarrow \quad p\langle \sigma, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel n(\!|z{:=}v|\!)}$$

Channel communication

- sending values + knowledge about σ

$$\frac{\neg closed(c_f[q_2]) \qquad \sigma' = \sigma}{p\langle\sigma, c \leftarrow v; t\rangle \parallel c[q_2] \quad \rightarrow \quad p\langle\sigma', t\rangle \parallel c[(v,\sigma) :: q_2]}\ \text{R-SEND}$$

$$\frac{v \neq \bot \qquad \sigma' = \sigma + \sigma''}{\begin{array}{rlll} c_b[q_1] \parallel & p\langle\sigma, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle & \parallel c_f[q_2 :: (v,\sigma'')] & \rightarrow \\ c_b[\sigma :: q_1] \parallel & p\langle\sigma', \texttt{let}\ r = v\ \texttt{in}\ t\rangle & \parallel c_f[q_2] & \end{array}}\ \text{R-REC}$$

Bounded channels

- A *send* on a channel happens-before the corresponding *receive* from that channel completes.
- The ith *receive* on a channel with capacity k happens-before the $i+k$th *send* from that channel completes.

Bounded channels

- A *send* on a channel happens-before the corresponding *receive* from that channel completes.
- The ith *receive* on a channel with capacity k happens-before the $i+k$th *send* from that channel completes.

Bounded channels

. There is also a "backward synchronization"

- from an "earlier" receive to a sender

Rest <2>

- *forward channel* (as shown)
- *backward* channel, propagating local σ knowledge

Send and receive

$$\frac{\neg closed(c_f[q_2]) \qquad \sigma' = \sigma + \sigma''}{c_b[q_1 :: \sigma''] \parallel p\langle \sigma, c \leftarrow v; t\rangle \parallel c_f[q_2] \quad \rightarrow \quad c_b[q_1] \parallel p\langle \sigma', t\rangle \parallel c_f[(v, \sigma) :: q_2]}\ \textsc{R-Send}$$

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

Delayed reads

- so far: delayed or buffered writes
- also *delayed reads* (load buffers) possible ⇒ "read events"

More (and more complex) "events"

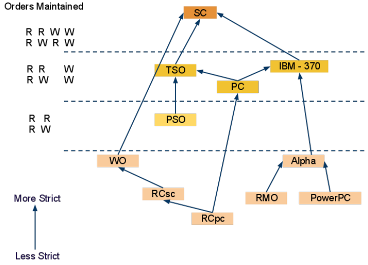
$$m(\!|\sigma, z := n_1|\!)_p \quad \text{and} \quad m[\![\sigma, ?n_4]\!]_p$$

Rest

- chain of "future references"
- symbolic execution
- nota bene:
  - the write itself is not "delayed"
  - it's the negative information (invisibility of other writes via the shadow sets, that travels slow)

Memory models

- WMMs, it's a jungle
- out-of-thin air
    - should be avoided (or should it?)
    - not even crystal clear what it is.



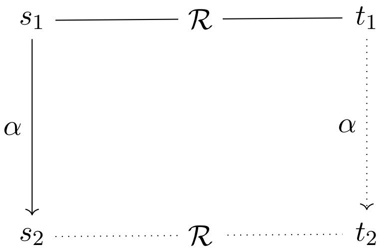
. . . but there's a bottom line

No matter how "relaxed" you want your memory model one thing is non-negotiable:

DRF-SF

Data-race free programs have to be sequentially consistent Manson et al. [8]

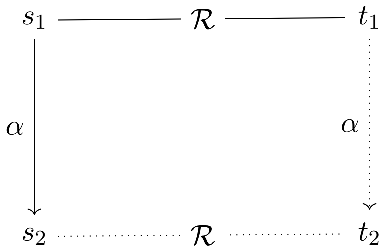
Simulation



"weak simulates strong"

sure thing

Simulation



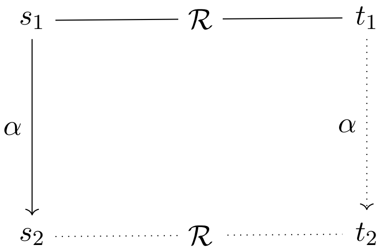
“weak simulates strong”

sure thing

“strong simulates weak”

definitely not

Simulation



"weak simulates strong"	"strong simulates weak"
sure thing	conditionally, for RF programs

Races

- “simultaneous” access to a shared location, where at least one is a write access

Manifest race (case W/W)

config C with

$$C \xrightarrow{p_1\langle z!\rangle}_s \xrightarrow{p_2\langle z!\rangle}_s$$

Rest

- race: reachable configuration with manifest race

Core of the proof

abstaction function/relation

"weak config" → "strong config"

Rest

- problem: configs contains "alternatives"

$$n_1(\!|z{:=}v_1|\!) \parallel n_2(\!|z{:=}v_2|\!)$$

- strong semantics: exactly one value of z

From local to global view ⇒ consensus

Lemma (Consensus possible)

Weak configurations obey the following invariant

$$\bigcap_{p \in P} W_P^{\circ}(z@p) \neq \emptyset \ . \tag{2}$$

- adding also *read events* to configurations

RF programs $\Rightarrow$ stronger consensus

Lemma (Race-free consensus when it counts)

Assume $P_0 \rightarrow_w^* P$ *with* P_0 *race-free. If* $P \xrightarrow{p(z?)}_w$ *or* $P \xrightarrow{p(z!)}_w$, *then*

$$\bigcap_{p_i} W_P^{\circ}(z@p_i) = \{n\} \ , \tag{3}$$

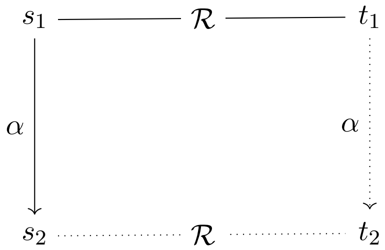
where the intersection ranges over an arbitrary set of processes which includes p.

Lemma (Race-free consensus)

Weak configurations for race-free programs obey the following invariant

$$\bigcap_{p_i \in P} W_P^{\circ}(z@p_i) = \{n\} \ . \tag{4}$$

Conditional simulation



- augment the configuration with additional read-events

$\Rightarrow$ consensus lemmas

$\Rightarrow$ DRF-SC

$\mathbb{K}$-Framework

- rewrite-based engine
- used variously for excecutable semantics
  - C++ memory models
  - "Etherium" smart contracts platform
  - ...
- see `https://github.com/dfava/mmgo`

Receive in $\mathbb{K}$

R-Rec

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

Rest

```
rule <goroutine>
      <k> <- channel(Ref:Int) => V ... </k>
      <sigma>
        <HB> HMap:Map => mergeHB(HMap, HMapDP) </HB>
        <S>  SSet:Set => SSet SSetDP </S>
      </sigma>
      <id> _ </id>
    </goroutine>
    <chan>
     <ref> Ref </ref>
     <type> _ </type>
     <forward> ListItem( ListItem(V)
               ListItem(HMapDP)
               ListItem(SSetDP) ) => .List </forward>
     <backward> BQ:List => ListItem( ListItem(HMap)
                           ListItem(SSet)) BQ </backward>
    </chan>
    requires notBool( V ==K $eot )
```

Related work

- loads of material on *axiomatic* semantics
- operational:
  - Boudol and Petri based on *rewriting theory*
  - Kang et al.: "promising" semantics with *"clocks"*
  - Flanagan and Freund: *adversarial* memory
  - Demange et al. Plan B (Java buffered write semantics BMM)
  - Pichon-Pharabod and Sewell: operational semantics avoiding OOTA
  - Alrahman et al.
  - Matthias Perner et al: parametrized semantics for NI (earlier today)
  - ...

Conclusion

- formalizing WMM for some calculus with channel
- DRF-SC simulation proof
- read delays under work

References I

Bibliography

[1] Alrahman, Y. A., Andric, M., Beggiato, A., and Lluch-Lafuente, A. (2014). Can we efficiently check concurrent programs under relaxed memory models in Maude? In Escobar, S., editor, *Rewriting Logic and Its Applications – 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *Lecture Notes in Computer Science*, pages 21–41. Springer Verlag.

[2] Boudol, G. and Petri, G. (2009). Relaxed memory models: An operational approach. In *Proceedings of POPL '09*, pages 392–403. ACM.

[3] Demange, D., Laporte, V., Zhao, L., Jagannathan, S., Pichardie, D., and Vitek, J. (2013). Plan B: A buffered memory model for Java. In *Proceedings of POPL '13*, pages 329–342. ACM.

[4] Flanagan, C. and Freund, S. N. (2010). Adversarial memory for detecting destructive races. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[5] Kang, J., Hur, C., Lahav, O., Vafeiadis, V., and Dreyer, D. (2017). A promising semantics for relaxed-memory concurrency. In Castagna, G. and Gordon, A. D., editors, *Proceedings of POPL '17*, pages 175–189. ACM.

[6] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[7] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691.

[8] Manson, J., Pugh, W., and Adve, S. V. (2005). The Java memory memory. In *Proceedings of POPL '05*. ACM.

[9] Pichon-Pharabod, J. and Sewell, P. (2016). A concurrency-semantics for relaxed atomics that permits optimisation and avoids out-of-thin-air executions. In *Proceedings of POPL '16*. ACM.