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 \( K \) rewriting framework
Memory model

**MCM**

A specification what to expect from a shared memory, 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.

![Diagram showing thread interactions]
How to specify a MM

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

\[
\begin{align*}
\text{T0} & \\
\text{a: } Wx &= 1 \\
\text{ppo} & \quad \text{fr} \\
\text{T1} & \\
\text{b: } Wy &= 1 \\
\end{align*}
\]

\[
\begin{align*}
\text{T0} & \\
\text{c: } Ry &= 1 \\
\text{rf} & \quad \text{ppo} \\
\text{T1} & \\
\text{d: } Rx &= 0 \\
\end{align*}
\]
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 \( \pi \) ...)
  - bounded channels
  - mixed choice
    - with “channel guards”
    - default-clause
Syntax

\[ v ::= r \mid n \]  \hspace{1cm} \text{values}

\[ e ::= t \mid v \mid \text{load } z \mid z := v \mid \text{if } v \text{ then } t \text{ else } t \mid \text{go } t \]  \hspace{1cm} \text{expression}

\[ \mid \text{make (chan } T, v) \mid v \leftarrow v \mid v \leftarrow v \mid \text{close } v \]

\[ g ::= v \leftarrow v \mid v \leftarrow v \mid \text{default} \]  \hspace{1cm} \text{guards}

\[ t ::= \text{let } r = e \text{ in } t \mid \sum_i \text{let } r_i = g_i \text{ in } t_i \]  \hspace{1cm} \text{threads}

- \[\sum : \text{choice (select, case, default)}\]
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
- straightforward 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: 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 \text{INVISIBLE}

\(^1\)independent from shared memory
Two(*) ingredients for HB only

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

**Sends and receives**

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

\[ P ::= n\langle \sigma, t \rangle \mid n(z := v) \mid \bullet \mid P \parallel P \mid n[q] \mid \nu n P. \]
Operational semantics (weak)

Configuration

\[ P ::= n\langle \sigma, t \rangle \mid n(z:=v) \mid \bullet \mid P \parallel P \mid n[q] \mid \nu n \, P. \] (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

\[ P ::= n\langle \sigma, t \rangle \mid n\langle z:=v \rangle \mid \bullet \mid P \parallel P \mid n[q] \mid \nu n \ P . \] (1)

<table>
<thead>
<tr>
<th>thread</th>
<th>write event</th>
<th>channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n\langle \sigma, t \rangle )</td>
<td>( n\langle z:=v \rangle )</td>
<td>( n[q] )</td>
</tr>
</tbody>
</table>
Write & read steps

\[ \sigma, z := v; t \rightarrow \sigma', t \parallel n(z := v) \]

\[ \sigma, \text{let } r = \text{load } z \text{ in } t \parallel n(z := v) \rightarrow \sigma, \text{let } r = v \text{ in } t \parallel n(z := v) \]
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 to 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**

\[ n(\sigma, t) \]

- local “state” tuple \((E_{hb}, E_s)\), \(\sigma : 2^{(N \times X)} \times 2^N\).
of course: shadow sets used to make writes invisible
local update of “program order”

\[
\begin{align*}
\sigma &= (E_{hb}, E_s) & \sigma' &= (E_{hb} + (n, z), E_s + E_{hb}(z)) \\
\rho(\sigma, z := v; t) &\rightarrow \rho(\sigma', t) \parallel n(\mid z := v) \\
\end{align*}
\]

\[
\begin{align*}
\sigma &= (_, E_s) & n \notin E_s \\
\rho(\sigma, \text{let } r = \text{load } z \text{ in } t) \parallel n(\mid z := v) &\rightarrow \rho(\sigma, \text{let } r = v \text{ in } t) \parallel n(\mid z := v)
\end{align*}
\]
Channel communication

- sending values + knowledge about $\sigma$

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

\[
v \neq \bot \quad \sigma' = \sigma + \sigma'' \\
c_b[q_1] \parallel p\langle \sigma, \text{let } r = \leftarrow c \text{ in } t \rangle \parallel c_f[q_2 :: (v, \sigma'')] \quad \rightarrow \\
c_b[\sigma :: q_1] \parallel p\langle \sigma', \text{let } r = v \text{ in } t \rangle \parallel c_f[q_2]
\]
Bounded channels

- A *send* on a channel happens-before the corresponding *receive* from that channel completes.
- The $i$th *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 $i$th *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 $\sigma$ knowledge
Send and receive

\[ \neg \text{closed}(c_f[q_2]) \quad \sigma' = \sigma + \sigma'' \]

\[ c_b[q_1 :: \sigma''] \parallel p(\sigma, c \leftarrow v; t) \parallel c_f[q_2] \quad \rightarrow \quad c_b[q_1] \parallel p(\sigma', t) \parallel c_f[(v, \sigma) :: q_2] \]

\[ v \neq \bot \quad \sigma' = \sigma + \sigma'' \]

R-SEND

\[ c_b[q_1] \parallel p(\sigma, \text{let } r \leftarrow c \text{ in } t) \parallel c_f[q_2 :: (v, \sigma'')] \quad \rightarrow \]

\[ c_b[\sigma :: q_1] \parallel p(\sigma', \text{let } r = v \text{ in } t) \parallel c_f[q_2] \]

R-REC
Delayed reads

- so far: delayed or buffered writes
- also delayed reads (load buffers) possible $\Rightarrow$ “read events”

More (and more complex) “events”

$$m(\sigma, z := n_1)_p \text{ and } m(\sigma, ?n_4)_p$$

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**

```
\begin{equation}
\begin{array}{c}
s_1 \stackrel{R}{\rightarrow} t_1 \\
\alpha \\
s_2 \stackrel{R}{\rightarrow} t_2
\end{array}
\end{equation}
```

"weak simulates strong"

"strong simulates weak"

**sure thing**

**definitely not**
Simulation

\[ s_1 \xrightarrow{\mathcal{R}} t_1 \]
\[ s_2 \xrightarrow{\mathcal{R}} t_2 \]

“weak simulates strong”

“strong simulates weak”

sure thing

conditionally, for RF programs
“simultaneous” access to a shared location, where at least one is a write access

**Manifest race (case W/W)**

config $C'$ with

$$C \xrightarrow{p_1(z!)} s \xrightarrow{p_2(z!)} s$$

**Rest**

- race: reachable configuration with manifest race
Core of the proof

abstraction function/relation

“weak config” → “strong config”

Rest

- problem: configs contains “alternatives”

\[ n_1(z \leftarrow v_1) \parallel n_2(z \leftarrow v_2) \]

- strong semantics: exactly one value of \( z \)
Lemma (Consensus possible)

Weak configurations obey the following invariant

\[ \bigcap_{p \in P} W_P(z \circ p) \neq \emptyset. \]  \( (2) \)

- 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

$$\bigcap_{p_i} W^o_P(z@p_i) = \{n\}, \quad (3)$$

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

$$\bigcap_{p_i \in P} W^o_P(z@p_i) = \{n\}. \quad (4)$$
Conditional simulation

- augment the configuration with additional read-events
  ⇒ consensus lemmas
  ⇒ DRF-SC
K-Framework

- rewrite-based engine
- used variously for executable semantics
  - C++ memory models
  - “Etherium” smart contracts platform
  - ...
- see https://github.com/dfava/mmgo
Receive in \( K \)

**R-Rec**

\[
v \neq \bot \quad \sigma' = \sigma + \sigma''
\]

\[
c_b[q_1] \parallel p\langle \sigma, \text{let } r = c \text{ in } t \rangle \parallel c_f[q_2 :: (v, \sigma'')] \rightarrow
c_b[\sigma :: q_1] \parallel p\langle \sigma', \text{let } r = v \text{ in } t \rangle \parallel c_f[q_2]
\]

**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) 
ListItm(HMapDP) 
ListItm(SSetDP)) => .List </\forward>

<\backward> BQ:List => ListItem( ListItem(HMap) 
ListItm(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


