Verifying concurrent C programs in Coq

Jean-Marie Madiot
joint work with Santiago Cuellar, Andrew Appel

Princeton University

Gallium seminar, January 4, 2016
Top to bottom verified software development

Verified software development:
- from top: specifications, program logics, static analysers
- to bottom: models of low-level architectures.

Existing tools perform well:
- reasoning: powerful program logics and analysers,
- translations: CompCert certified compiler,
- models of weak memory for different architectures.

The Verified Software Toolchain (VST) project in Princeton can already verify complex C programs in Coq.

https://vst.cs.princeton.edu/

This talk: we try and extend VST to concurrent C programs.
Verified software development:

- from top: specifications, program logics, static analysers
- to bottom: models of low-level architectures.

Existing tools perform well:

- reasoning: powerful program logics and analysers,
- translations: CompCert certified compiler,
- models of weak memory for different architectures.

The Verified Software Toolchain (VST) project in Princeton can already verify complex C programs in Coq.

[https://vst.cs.princeton.edu/](https://vst.cs.princeton.edu/)

This talk: we try and extend VST to concurrent C programs.
Top to bottom verified software development

Verified software development:
- from top: specifications, program logics, static analysers
- to bottom: models of low-level architectures.

Existing tools perform well:
- reasoning: powerful program logics and analysers,
- translations: CompCert certified compiler,
- models of weak memory for different architectures.

The Verified Software Toolchain (VST) project in Princeton can already verify complex C programs in Coq.

https://vst.cs.princeton.edu/
Top to bottom verified software development

Verified software development:

- from top: specifications, program logics, static analysers
- to bottom: models of low-level architectures.

Existing tools perform well:

- reasoning: powerful program logics and analysers,
- translations: CompCert certified compiler,
- models of weak memory for different architectures.

The Verified Software Toolchain (VST) project in Princeton can already verify complex C programs in Coq.

https://vst.cs.princeton.edu/

This talk: we try and extend VST to concurrent C programs.
Architecture of VST

Contributors:
Andrew Appel
Lennart Beringer
Aquinas Hobor (PhD ’08)
Robert Dockings (PhD ’12)
Gordon Stewart (PhD ’15)
Joey Dodds (PhD ’15)
Qinxiang Cao (grad student)
Santiago Cuellar (grad student)
Nick Giannarakis (grad student)
Jean-Marie Madiot (postdoc)
Architecture of VST

- CompCert : C program $\rightarrow$ Power PC code: preserves the semantics
- VST’s separation logic: predicates on this semantics,
- VST’s program logic: functional correctness of such programs.
VST’s higher-order separation logic

\[ P ::= \ldots \quad v \downarrow 4 \quad \text{local variables} \]
\[ p \mapsto 4, p \mapsto \_ \quad \text{pointers, shape} \]
\[ f : P \rightarrow Q \quad \text{function pointers (indirection)} \]
\[ !!P \quad \text{embedding of Coq propositions} \]
\[ \land, \lor \quad \text{usual logical operators} \]
\[ \mu x. P \quad \text{recursion} \]
\[ \forall x. P, \exists x. P \quad \text{impredicative quantification} \]
\[ P * Q, P \rightarrow Q \quad \text{non-aliasing (separation)} \]

Recent work was necessary to handle all those features:
Step indexing (Appel, McAllester, TOPLAS 2001)
Step indexing + indirection (Ahmed, Appel, Virga, LICS 2002)
Step indexing + impredicativity (Ahmed PhD thesis 2004)
Very Modal Model (Appel, Melliès, Richards, Vouillon, POPL 2007)
Indirection Theory (Hobor, Dockins, Appel, POPL 2010)
Example of a proof of a program
Example of a C program

```c
struct list {int head; struct list *tail;};

struct list *merge(struct list *a, struct list *b) {
    struct list* ret;
    struct list** x = &ret;
    while (a && b) {
        if (a->head <= b->head) {
            *x = a;
            a = a->tail;
        } else {
            *x = b;
            b = b->tail;
        }
        x = &((*x)->tail);
    }
    *x = (a)?a:b;
    return ret;
}
```

To notice: addressable local variables, pointer to undefined values, loop invariant with partially defined list segments, pointer tricks, no leak.
Same program in \textit{verifiable} C

```c
#include <stdio.h>

struct list {int head; struct list *tail};

struct list *merge(struct list *a, struct list *b) {
    struct list* ret;
    struct list* temp;
    struct list** x;
    int va, vb, cond;
    x = &ret;
    cond = a != NULL && b != NULL;
    while (cond) {
        va = a->head;
        vb = b->head;
        if (va <= vb) {
            *x = a;
            x = &(a->tail);
            a = a->tail;
        } else {
            *x = b;
            x = &(b->tail);
            b = b->tail;
        }
        cond = a != NULL && b != NULL;
    }
    if (a != NULL) {
        *x = a;
    } else {
        *x = b;
    }
temp = ret;
return temp;
}
```

Transformation to \textit{verifiable} C:

- **temp**: addressable variables can’t be returned
- **va, vb**: tests can be on local expressions only
- **cond**: tests can’t be transformed in instructions
- loads and stores must be top-level (which forbids $x = &(*x)->\text{tail}$)

(the transformation could be done automatically, but we still need to reason on this program)
Proof of merge.c

```
reveal exists_pre for us.
rename a into init_a.
rename b into init_b.
clear a, b.
intros cond a b merged a_b c begin.
forward.

(* The loop *)
forward_while (merge_invariant _cond sh init_a init_b ret)
[[[[[[cond0 a]0 b]0 merged0] a]0 b]0 c]0 begin0].
+ (* Loop: precondition => invariant *)
  Exists cond a b merged a_b c begin; entail!.!
+ (* Loop: condition has nice format *)
now entail!.
+ (* Loop body preserves invariant *)
clear - SH HRE H1 H2.
rename cond0 into cond, a0 into a, b0 into b, merged0 into merged,
a0 into a, b0 into b, c0 into c, begin0 into begin.
assert (a <= nullval) by intuition.
assert (b <= nullval) by intuition.

clear H2.
drop LOCAL 4_nat; clear cond HRE.
rewrite lseg_unfold.
destruct a as [va a']; simpl.
  (* [a] cannot be empty *)
now intuition.

normalize.
intros a,.'.
normalize.
  (* Now the command [va = a->head] can proceed *)
rewrite list_cell_field_at.
forward.

rewrite lseg_unfold with (vl:=b_).
destruct b as [vb b']; simpl.
  (* [b] cannot be empty *)
now intuition.
normalize.
intros b,.'.
normalize.
clear H2 H3.

---------
```

```
H1: merge init_a init_b = merged ++ merge a b
H2: cond = Int.zero <-> a_ = nullval \ b_ = nullval
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS := abbreviate : statement
H: a <<= nullval
H0: b <<= nullval

semax Delta
(PROP ()
  LOCAL (temp_a a_; temp_b b_;
    temp_x,
      if merged
        then ret_
          else field_address (Tstruct_list noattr) [StructField_tail c_];
    lvar _ret list_ret; temp_cond (Vint cond))
  SEP ('lseg LS (map Vint a_ a_nullval);
    'lseg LS sh (map Vint (but last merged) begin c_);
    (if merged
      then emp
      else data at sh t_struct_list (Vint (last merged), Vundef c_)))
  (Sequence
    (Sset va
      (Efield
        (Ederefer Ettempvar _a (tptr (Tstruct_list noattr)))
          (Tstruct_list noattr) _head tint))
    MORE_COMMANDS)

POSTCONDITION

```

U:%-  *goals*  Bot (36,50)  (Coq Goals -2 vl

---
```

Jean-Marie Madiot (Princeton)  Verifying concurrent C programs in Coq  Gallium, January 4, 2016  7 / 32
Concurrent programs
Simple concurrent program

```c
x = 0;
y = 0;
x++ || y++;
assert(x + y == 2);
```
Proof of simple concurrent program

\[
\begin{align*}
\{ x \mapsto \_ \ast y \mapsto \_ \} \\
x &= 0; \\
\{ x \mapsto 0 \ast y \mapsto \_ \} \\
y &= 0; \\
\{ x \mapsto 0 \ast y \mapsto 0 \} \\
x++ \, || \, y++; \\
(*) \quad \{ x \mapsto 1 \ast y \mapsto 1 \} \\
\text{assert}(x + y == 2); \\
\{ x \mapsto 1 \ast y \mapsto 1 \}
\end{align*}
\]
Proof of simple concurrent program

\[
\{ x \mapsto \ast y \mapsto \_ \}
\]

\[
x = 0; \\
x \mapsto \_ * y \mapsto \_
\]

\[
y = 0; \\
y \mapsto \_ * y \mapsto 0
\]

\[
x++ \ || \ y++; \\
\{ x \mapsto 1 * y \mapsto 1 \}
\]

\[
(\ast) \quad \{ x \mapsto 0 \} \quad \{ x \mapsto 1 \}
\]

\[
\{ x \mapsto 0 * y \mapsto 0 \} \quad \{ y \mapsto 0 \} \quad \{ y \mapsto 1 \}
\]

\[
x++ \ || \ y++; \\
\{ x \mapsto 1 * y \mapsto 1 \}
\]

\[
\{ x \mapsto 1 * y \mapsto 1 \}
\]


(no “no interference”)
Proof of simple concurrent program

\[
x = 0; \ y = 0;
x++ \ || \ y++;
assert(x + y == 2);
\]

The program above is safe.
Proof of simple concurrent program

```
x = 0; y = 0;
x++ || y++;
assert(x + y == 2);
```

The program above is safe.

But we have no shared resources.
Threads sharing memory

x = 0;
x++ || x++;
assert(x >= 0);
Threads sharing memory

x = 0;
x++ || x++;
assert(x >= 0);

...race?
Races in CompCert

There are **no** data races in CompCert/VST:

- most experimental logic with races are not proved sound for weakly consistent caches;

Concurrent variants of CompCert:

- [CompCert TSO, Sewell] racy programs with little ability for the compiler to optimize
- [Compositional CompCert, Appel/Beringer/Stewart/Cuellar] coarse-grain concurrency and optimizing compilation of memory operations
Races in CompCert

There are no data races in CompCert/VST:

- most experimental logic with races are not proved sound for weakly consistent caches;
- our program logic ensures the absence of race;
There are no data races in CompCert/VST:

- most experimental logic with races are not proved sound for weakly consistent caches;
- our program logic ensures the absence of race;
- CompCert 2.0’s semantics gets stuck at racy loads and stores, using a permission model.
Races in CompCert

There are no data races in CompCert/VST:

- most experimental logic with races are not proved sound for weakly consistent caches;
- our program logic ensures the absence of race;
- CompCert 2.0’s semantics gets stuck at racy loads and stores, using a permission model.

Concurrent variants of CompCert:

- either [CompCert TSO, Sewell] racy programs with little ability for the compiler to optimize
Races in CompCert

There are no data races in CompCert/VST:
- most experimental logic with races are not proved sound for weakly consistent caches;
- our program logic ensures the absence of race;
- CompCert 2.0’s semantics gets stuck at racy loads and stores, using a permission model.

Concurrent variants of CompCert:
- either [CompCert TSO, Sewell] racy programs with little ability for the compiler to optimize
- or [Compositional CompCert, Appel/Beringer/Stewart/Cuellar] coarse-grain concurrency and optimizing compilation of memory operations
Permissions in CompCert

CompCert memory model, version 2:

\[ x \xrightarrow{\pi} 4 \quad \text{rather than} \quad x \mapsto 4 \]

\[ \pi ::= \text{Freeable} \mid \text{Writable} \mid \text{Readable} \mid \text{Nonempty} \]
Permissions in CompCert

CompCert memory model, version 2:

\[ x \overset{\pi}{\mapsto} 4 \quad \text{rather than} \quad x \mapsto 4 \]

\[ \pi ::= \text{Freeable} > \text{Writable} > \text{Readable} > \text{Nonempty} \]
Permissions in CompCert

CompCert memory model, version 2:

\[ x \xrightarrow{\pi} 4 \quad \text{rather than} \quad x \xrightarrow{} 4 \]

\[ \pi ::= \text{Freeable} > \text{Writable} > \text{Readable} > \text{Nonempty} \]

- if a thread has Freeable, others have no permissions;
- if a thread has Writable, others have at most Nonempty;
- if a thread has Readable, others have at most Readable;
- if a thread has Nonempty, others have at most Writable.
Permissions in CompCert

CompCert memory model, version 2:

\[ x \mapsto 4 \quad \text{rather than} \quad x \mapsto 4 \]

\[ \pi ::= \text{Freeable} > \text{Writable} > \text{Readable} > \text{Nonempty} \]

- if a thread has Freeable, others have no permissions;
- if a thread has Writable, others have at most Nonempty;
- if a thread has Readable, others have at most Readable;
- if a thread has Nonempty, others have at most Writable.

Nonempty is “comparable with another non-NULL pointer”: in \( a == b \), if one of \( a \) or \( b \) is a pointer value, then either one of them must be NULL, or both must be pointers to allocated objects (Nonempty ensures there are no other Freeable).
Permissions in VST

Refinement of CompCert’s permissions:

\[ \pi ::= \cdot \mid \circ \mid \pi_1 \pi_2 \]

Joining permissions:

\[ \begin{align*}
\cdot \circ \oplus \circ \bullet &= \bullet \bullet = \bullet \\
\pi = \pi_1 \oplus \pi_2 \quad &\quad p \overset{\pi}{\rightarrow} v = p \overset{\pi_1}{\rightarrow} v \ast p \overset{\pi_2}{\rightarrow} v
\end{align*} \]

Embedding, depending on where the $\bullet$s are:
Threads sharing memory

x = 0;
x++ || x++;
assert(x >= 0);

...race?
Threads sharing memory, using binary semaphores

\begin{align*}
x &= 0; \\
V(s); \\
P(s); &\parallel P(s); \\
x++; &\parallel x++; \\
V(s); &\parallel V(s); \\
P(s); \\
\text{assert}(x \geq 0);
\end{align*}

no race!
#include <pthread.h>
#include <semaphore.h>

void assert(int i) { i = 1/i; }

sem_t s;
int x;

int main (void) {
    pthread_t th;
    x = 0;
    sem_init(&s, 0, 0);
    sem_post(&s);
    pthread_create(&th, NULL, f, (void*)&x);
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_join(th, NULL);
    sem_wait(&s);
    sem_destroy(&s);
    assert(x >= 0);
    return 0;
}

void* f(void *arg) {
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_exit(NULL);
}
Binary semaphores contain permissions, here on $x$, which can be transferred between threads:

\[
\begin{align*}
P(s); & \quad P(s); \\
++x; & \quad ++x; \\
V(s); & \quad V(s);
\end{align*}
\]
Threads sharing memory, using binary semaphores

Binary semaphores contain permissions, here on \( x \), which can be transferred between threads:

\[
\begin{align*}
\{ s \leftrightarrow \text{lock}[x] \} \\
P(s) ; \\
\{ s \leftrightarrow \text{lock}[x] \ast x \mapsto _{-} \}
\end{align*}
\]

\[
\begin{align*}
\{ s \leftrightarrow \text{lock}[x] \ast x \mapsto _{-} \}
\end{align*}
\]

\[
\begin{align*}
x++ ; \\
\{ s \leftrightarrow \text{lock}[x] \ast x \mapsto _{-} \}
\end{align*}
\]

\[
\begin{align*}
\{ s \leftrightarrow \text{lock}[x] \}
\end{align*}
\]

\[
\begin{align*}
V(s) ; \\
\{ s \leftrightarrow \text{lock}[x] \}
\end{align*}
\]
Threads sharing memory

Permissions can be refined to *lock invariants*:

\[
\begin{align*}
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n]\} \\
\text{P(s);} \\
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast \exists n \geq 0 \ x \mapsto n\} \\
x++; \\
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n]\} \\
\text{V(s);} \\
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n]\}
\end{align*}
\]

\[
\begin{align*}
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n]\} \\
\text{P(s);} \\
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast \exists n \geq 0 \ x \mapsto n\} \\
x++; \\
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n]\} \\
\text{V(s);} \\
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n]\}
\end{align*}
\]
Threads sharing memory

Atomicity is not mandatory:

\[
\{ s \mapsto \text{lock} \left[ \exists n \geq 0 \ x \mapsto n \right] \} \\
\text{P}(s); \\
\{ s \mapsto \text{lock} \left[ \exists n \geq 0 \ x \mapsto n \right] \} \ast \{ \exists n \geq 0 \ x \mapsto n \} \\
a = x; \\
x = a - 2; // x can be negative here \\
x = a + 1; \\
\{ s \mapsto \text{lock} \left[ \exists n \geq 0 \ x \mapsto n \right] \} \ast \{ \exists n \geq 0 \ x \mapsto n \} \\
\text{V}(s); \\
\{ s \mapsto \text{lock} \left[ \exists n \geq 0 \ x \mapsto n \right] \} \\
\]
Threads sharing memory

Coming back to $x++$:

$$\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \}$$

$P(s);$  
$$\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \} \ast \exists n \geq 0 \ x \mapsto n \}$$

$x++;$  
$$\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \} \ast \exists n \geq 0 \ x \mapsto n \}$$

$V(s);$  
$$\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \}$$

$$\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \}$$

$P(s);$  
$$\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \} \ast \exists n \geq 0 \ x \mapsto n \}$$

$x++;$  
$$\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \} \ast \exists n \geq 0 \ x \mapsto n \}$$

$V(s);$  
$$\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \}$$
Threads sharing memory

\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast x \mapsto \_ \}\}

x = 0;
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast x \mapsto 0 \}\}
V(s);
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \}\}
(P(s); \ x++; \ V(s)) \parallel (P(s); \ x++; \ V(s));
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \}\}
P(s);
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast \exists n \geq 0 \ x \mapsto n \}\}
assert(x \geq 0);
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast \exists n \geq 0 \ x \mapsto n \}\}

The above program is safe.

But we can know more about x

Jean-Marie Madiot (Princeton)
Verifying concurrent C programs in Coq
Gallium, January 4, 2016 20 / 32
Threads sharing memory

\[
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \} \ast x \mapsto \_ \}
\]

\[
x = 0;
\]

\[
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast x \mapsto 0 \}
\]

\[
V(s);
\]

\[
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \}
\]

\[
(P(s); \ x++; \ V(s)) \ || \ (P(s); \ x++; \ V(s));
\]

\[
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \}
\]

\[
P(s);
\]

\[
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast \exists n \geq 0 \ x \mapsto n \}
\]

\[
\text{assert}(x \geq 0);
\]

\[
\{ s \mapsto \text{lock}[\exists n \geq 0 \ x \mapsto n] \ast \exists n \geq 0 \ x \mapsto n \}
\]

The above program is safe.
Threads sharing memory

\[
x = 0; \\
V(s); \ \\
(P(s); x++; V(s)) \parallel (P(s); x++; V(s)); \\
P(s); \ \\
assert(x \geq 0);
\]

The above program is safe.
Threads sharing memory

```
x = 0;
V(s);
(P(s); x++; V(s)) || (P(s); x++; V(s));
P(s);
assert(x >= 0);
```

The above program is safe.

But we can know more about $x$
Threads sharing memory

x = 0;
V(s);
(P(s); x++; V(s)) || (P(s); x++; V(s));
P(s);
assert(x == 2);
Threads sharing memory

```c
x = 0;
V(s);
(P(s); x++; V(s)) || (P(s); x++; V(s));
P(s);
assert(x == 2);
```

*Invariants* are not enough.
Threads sharing memory: ghost variables

\[
x = 0; \quad x_1 = 0; \quad x_2 = 0;
\]
\[
V(s);
\]
\[
(P(s); \quad x_1++; \quad x++; \quad V(s)) \quad || \quad (P(s); \quad x_2++; \quad x++; \quad V(s));
\]
\[
P(s);
\]
\[
assert(x == 2);
\]
Ghost variables

A new invariant relying on ghost variables: \[ R = \exists n_1, n_2 \cdot x_1 \mapsto n_1 \]
\[ \cdot x_2 \mapsto n_2 \]

\[
\begin{align*}
\{ l \mapsto R \cdot x_2 \mapsto 0 \} \\
P(1); \\
\{ l \mapsto R \cdot \exists n_1 \cdot x_2 \mapsto 0 \cdot x \mapsto n_1 + 0 \cdot x_1 \mapsto n_1 \} \\
\{ l \mapsto R \cdot x_2 \mapsto 0 \cdot x \mapsto n_1 + 0 \cdot x_1 \mapsto n_1 \} \\
x_2++; \\
\{ l \mapsto R \cdot x_2 \mapsto 1 \cdot x \mapsto n_1 + 0 \cdot x_1 \mapsto n_1 \} \\
x++; \\
\{ l \mapsto R \cdot x_2 \mapsto 1 \cdot x \mapsto n_1 + 1 \cdot x_1 \mapsto n_1 \} \\
\{ l \mapsto R \cdot x_2 \mapsto 1 \cdot R \} \\
V(1); \\
\{ l \mapsto R \cdot x_2 \mapsto 1 \} 
\end{align*}
\]
Threads sharing memory

\{ s \xrightarrow{\square} R \cdot x \xrightarrow{\bullet} _{\bullet} \}\n
x = 0; \ x1 = 0; \ x2 = 0;
\{ s \xrightarrow{\square} R \cdot x \xrightarrow{\bullet} 0 \cdot x1 \xrightarrow{\bullet} 0 \cdot x2 \xrightarrow{\bullet} 0 \}\nV(s);
\{ s \xrightarrow{\square} R \cdot x1 \xrightarrow{\bullet} 0 \cdot x2 \xrightarrow{\bullet} 0 \}\n(P(s); x1++; x++; V(s)) || (P(s); x2++; x++; V(s));
\{ s \xrightarrow{\square} R \cdot x1 \xrightarrow{\bullet} 1 \cdot x2 \xrightarrow{\bullet} 1 \}\nP(s);
\{ s \xrightarrow{\square} R \cdot x \xrightarrow{\bullet} 2 \cdot x1 \xrightarrow{\bullet} 1 \cdot x2 \xrightarrow{\bullet} 1 \}\nassert(x == 2);
\{ s \xrightarrow{\square} R \cdot x \xrightarrow{\bullet} 2 \cdot x1 \xrightarrow{\bullet} 1 \cdot x2 \xrightarrow{\bullet} 1 \}
Threads sharing memory

\{ s \rightarrow R \ast x \rightarrow \_ \} \\
\begin{align*}
x &= 0; \ x_1 &= 0; \ x_2 &= 0; \\
\{ s \rightarrow R \ast x \rightarrow 0 \ast x_1 \rightarrow 0 \ast x_2 \rightarrow 0 \} \\
V(s); \\
\{ s \rightarrow R \ast x_1 \rightarrow 0 \ast x_2 \rightarrow 0 \} \\
(P(s); \ x_1++; \ x++; \ V(s)) || (P(s); \ x_2++; \ x++; \ V(s)); \\
\{ s \rightarrow R \ast x_1 \rightarrow 1 \ast x_2 \rightarrow 1 \} \\
P(s); \\
\{ s \rightarrow R \ast x \rightarrow 2 \ast x_1 \rightarrow 1 \ast x_2 \rightarrow 1 \} \\
assert(x == 2); \\
\{ s \rightarrow R \ast x \rightarrow 2 \ast x_1 \rightarrow 1 \ast x_2 \rightarrow 1 \} \\
\end{align*}

The above program is safe.
But

Problems:

- unbounded number of ghost variables?
But

Problems:
- unbounded number of ghost variables, or thread flow unknown?

Importantly, when we own $g \mapsto v$ or $\exists v \ g \mapsto v$ we know that $v$ is not modified by another thread.
But

Problems:

- unbounded number of ghost variables, or thread flow unknown?
- erasure theorem on proofs in a shallow embedding?
But

Problems:
- unbounded number of ghost variables, or thread flow unknown?
- erasure theorem on proofs in a shallow embedding?
- ... isn’t it a logical problem?

Solution: we use an enriched memory (same as for $f$):

$\{\exists g \rightarrow v \neq g \rightarrow v\} \rightarrow \{Q\} \rightarrow \{P\} \rightarrow \{Q\} \rightarrow \{g \rightarrow v \neq g \rightarrow v\}$

Importantly, when we own $g \rightarrow v \neq g \rightarrow v$ or $\exists v \rightarrow g \rightarrow v \neq g \rightarrow v$ we know that $v$ is not modified by another thread.
But

Problems:
- unbounded number of ghost variables, or thread flow unknown?
- erasure theorem on proofs in a shallow embedding?
- ... isn’t it a logical problem?

Solution:
- we use a enriched memory (same as for $f : \{P\} \rightarrow \{Q\}$):

$$
\begin{align*}
\{\exists g \, g \mapsto v \ast P\} &\rightarrow c \{Q\} \\
\{P\} &\rightarrow c \{Q\} \\
\{g \mapsto v' \ast P\} &\rightarrow c \{Q\} \\
\{g \mapsto v \ast P\} &\rightarrow c \{Q\}
\end{align*}
$$

$$
\begin{align*}
g \mapsto v &= g \mapsto v \ast g \mapsto v
\end{align*}
$$

Importantly, when we own $g \mapsto v$ or $\exists \nu \, g \mapsto \nu$ we know that $\nu$ is not modified by another thread.
But

Problems:
- unbounded number of ghost variables, or thread flow unknown?
- erasure theorem on proofs in a shallow embedding?
- ... isn’t it a *logical* problem?

Solution:
- we use a *enriched* memory (same as for \( f : \{ P \} \to \{ Q \} \)):

\[
\begin{align*}
\{ \exists g \ g \mapsto^\bullet v \star P \} & \implies \{ Q \} \\
\{ P \} & \implies \{ Q \} \\
\{ g \mapsto^\bullet v' \star P \} & \implies \{ Q \} \\
\{ g \mapsto^\bullet v \star P \} & \implies \{ Q \}
\end{align*}
\]

\[
g \mapsto^\bullet v = g \mapsto^\bullet v \star g \mapsto^\bullet v
\]

- Importantly, when we own \( g \mapsto^\bullet v \) or \( \exists v \ g \mapsto^\bullet v \) we know that \( v \) is not modified by another thread.
But

Problems:

- unbounded number of ghost variables, or thread flow unknown?
- erasure theorem on proofs in a shallow embedding?
- ... isn’t it a logical problem?

Solution:

- we use a enriched memory (same as for $f : \{P\} \rightarrow \{Q\}$):

  $$\begin{align*}
  &\{\exists g \; g \mapsto v * P\} \; c \; \{Q\} \\
  \rightarrow \quad &\{P\} \; c \; \{Q\} \\
  \end{align*}$$

  $$\begin{align*}
  &\{g \mapsto v' * P\} \; c \; \{Q\} \\
  \rightarrow \quad &\{g \mapsto v * P\} \; c \; \{Q\} \\
  \end{align*}$$

  $$g \mapsto v \; = \; g \mapsto v \; * \; g \mapsto v$$

- Importantly, when we own $g \mapsto v$ or $\exists v \; g \mapsto v$ we know that $v$ is not modified by another thread.
- semantic erasure
But

Problems:
- unbounded number of ghost variables, or thread flow unknown?
- erasure theorem on proofs in a shallow embedding?
- ... isn’t it a *logical* problem?

Solution:
- we use a *enriched* memory (same as for \( f : \{P\} \rightarrow \{Q\} \)):

\[
\begin{align*}
\{\exists g \quad g \mapsto v \ast P\} & \quad c \quad \{Q\} \\
\{P\} & \quad c \quad \{Q\} \\
\{g \mapsto v \ast P\} & \quad c \quad \{Q\}
\end{align*}
\]

\[
\quad g \mapsto v = g \mapsto v \ast g \mapsto v
\]

- Importantly, when we own \( g \mapsto v \) or \( \exists v \quad g \mapsto v \) we know that \( v \) is not modified by another thread.
- *semantic* erasure
- infinite number of ghost variables?
But

Problems:
- unbounded number of ghost variables, or thread flow unknown?
- erasure theorem on proofs in a shallow embedding?
- ... isn’t it a logical problem?

Solution:
- we use a enriched memory (same as for \( f : \{P\} \to \{Q\} \)):

\[
\begin{align*}
\{\exists g \ g \mapsto v \ast P\} & \c c \{Q\} \\
\{P\} & \c c \{Q\}
\end{align*}
\begin{align*}
\{g \mapsto v' \ast P\} & \c c \{Q\} \\
\{g \mapsto v \ast P\} & \c c \{Q\}
\end{align*}
\]

\[
g \mapsto v = g \mapsto v \ast g \mapsto v
\]

- Importantly, when we own \( g \mapsto v \) or \( \exists v \ g \mapsto v \) we know that \( v \) is not modified by another thread.
- semantic erasure
- infinite number of ghost variables? indexed \( g_i \)'s?
But

Problems:
- unbounded number of ghost variables, or thread flow unknown?
- erasure theorem on proofs in a shallow embedding?
- ... isn’t it a logical problem?

Solution:
- we use a enriched memory (same as for $f : \{P\} \rightarrow \{Q\}$):

$$\begin{align*}
\{\exists g \; g \mapsto v \ast P\} \; c \; \{Q\} & \quad \frac{\{g \mapsto v' \ast P\} \; c \; \{Q\}}{\{P\} \; c \; \{Q\}} \\
\{g \mapsto v \ast P\} \; c \; \{Q\} & \quad \frac{\{g \mapsto v' \ast P\} \; c \; \{Q\}}{\{g \mapsto v \ast P\} \; c \; \{Q\}}
\end{align*}$$

$$g \mapsto v \; = \; g \mapsto v \ast g \mapsto v$$

- Importantly, when we own $g \mapsto v$ or $\exists v \; g \mapsto v$ we know that $v$ is not modified by another thread.
- **semantic** erasure
- infinite number of ghost variables? **indexed** $g_i$’s?

How to organise them? (we must keep an infinite supply!)
Splitting infinite sets

We can split infinite subsets, e.g. for $\mathbb{N}$:

$$\mathbb{N} = (1 + 2\mathbb{N}) \cup 2\mathbb{N}$$

and more that once:

$$\mathbb{N} = \bigcup_{k \in \mathbb{N}} 2^k (1 + 2\mathbb{N}) - 1$$
We have encountered this problem before!

Permissions shares have been implemented by $z, 0 \leq z \leq 1$, intervals of $[0, 1]$, subsets of $\mathbb{N}$, ... and finally, trees!
We have encountered this problem before!

Permissions shares have been implemented by $z$, $0 \leq z \leq 1$, intervals of $[0, 1]$, subsets of $\mathbb{N}$, ... and finally, trees!

$$t ::= \bullet \mid \circ \mid \overline{t_1 t_2} \mid \overline{\bullet \bullet} \equiv \bullet, \overline{\circ \circ} \equiv \circ$$
We have encountered this problem before!

Permissions shares have been implemented by $z$, $0 \leq z \leq 1$, intervals of $[0, 1]$, subsets of $\mathbb{N}$, ... and finally, trees!

$$t ::= \bullet \mid \circ \mid t_1 \bigwedge t_2 \quad / \quad \bullet \bullet \equiv \bullet, \quad \circ \circ \equiv \circ$$

Embedding in infinite-or-empty subsets of $\mathbb{N}$:

$$\mathbb{N}_\bullet = \mathbb{N} \quad \mathbb{N}_\circ = \emptyset \quad \mathbb{N}(t_1 \bigwedge t_2) = 2\mathbb{N}t_1 \cup (1 + 2\mathbb{N}t_2)$$
We have encountered this problem before!

Permissions shares have been implemented by $z, 0 \leq z \leq 1$, intervals of $[0, 1]$, subsets of $\mathbb{N}$, ... and finally, trees!

$$t ::= \bullet \mid \circ \mid t_1 \uparrow t_2 \quad / \quad \bullet \bullet \equiv \bullet, \quad \circ \circ \equiv \circ$$

Embedding in infinite-or-empty subsets of $\mathbb{N}$:

$$\mathbb{N}_\bullet = \mathbb{N} \quad \mathbb{N}_\circ = \emptyset \quad \mathbb{N}(t_1 \uparrow t_2) = 2\mathbb{N}_{t_1} \uplus (1 + 2\mathbb{N}_{t_2})$$

Converse ("terminates" on $[\cdot \cdot]$; $f \circ \mathbb{N}$. is the normalization function for $\equiv$)

$$f(\emptyset) = \circ \quad f(\mathbb{N}) = \bullet$$

$$f(A) = t_1 \uparrow t_2 \quad \text{with} \quad t_1 = f\left(\frac{A \cap 2\mathbb{N}}{2}\right) \quad \text{and} \quad t_2 = f\left(\frac{A \cap (1 + 2\mathbb{N}) - 1}{2}\right)$$
We have encountered this problem before!

Permissions shares have been implemented by $z, 0 \leq z \leq 1$, intervals of $[0, 1]$, subsets of $\mathbb{N}$, ... and finally, trees!

$$t ::= \bullet \mid \circ \mid t_1 \triangleleft t_2 \mid / \bullet \bullet \equiv \bullet, \circ \circ \equiv \circ$$

Embedding in infinite-or-empty subsets of $\mathbb{N}$:

$$\mathbb{N}_\bullet = \mathbb{N} \quad \mathbb{N}_\circ = \emptyset \quad \mathbb{N}(t_1 \triangleleft t_2) = 2\mathbb{N}t_1 \uplus (1 + 2\mathbb{N}t_2)$$

Converse ("terminates" on $[\cdot \cdot]$ ; $f \circ \mathbb{N}$. is the normalization function for $\equiv$)

$$f(\emptyset) = \circ \quad f(\mathbb{N}) = \bullet$$

$$f(A) = t_1 \triangleleft t_2 \quad \text{with} \quad t_1 = f \left( \frac{A \cap 2\mathbb{N}}{2} \right) \quad \text{and} \quad t_2 = f \left( \frac{A \cap (1 + 2\mathbb{N}) - 1}{2} \right)$$

These $\mathbb{N}_t$ help us embed our ghost state in our memory model.
Representation of ghost state

\[ g \xrightarrow{\pi} v \triangleq \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \xrightarrow{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v \]
Representation of ghost state

\[ g \xrightarrow{\pi} v \triangleq \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \xrightarrow{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v \]

Two tree shares:

- \( \pi \): permission (what can we do...)
- \( \rho \): location (...to which part)
Representation of ghost state

\[ g \xrightarrow{\pi \rho} v \triangleq \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \xrightarrow{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v \]

Two tree shares:
- \( \pi \): permission (what can we do...)
- \( \rho \): location (...to which part)

Composed value:
- \( v \) (the sum is finite)
  (can be any PCM)
Representation of ghost state

\[ g \xrightarrow[\rho]{\pi} v \triangleq \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \xrightarrow[\rho]{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v \]

Two tree shares:
- \( \pi \): permission (what can we do...)
- \( \rho \): location (…to which part)

Composed value:
- \( v \) (the sum is finite)
  (can be any PCM)

\[ \pi_1 \oplus \pi_2 = \pi \]

\[ g \xrightarrow[\rho]{\pi_1} v \ast g \xrightarrow[\rho]{\pi_2} v = g \xrightarrow[\rho]{\pi} v \]
Representation of ghost state

\[
g \xrightarrow{\rho} v \triangleq \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \xrightarrow{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v
\]

Two tree shares:
- \(\pi\): permission (what can we do...)
- \(\rho\): location (...to which part)

Composed value:
- \(v\) (the sum is finite)
  (can be any PCM)

\[
\rho_1 \oplus \rho_2 = \rho \\
v_1 \cdot v_2 = v
\]

\[
g \xrightarrow{\rho_1} v_1 \ast g \xrightarrow{\rho_2} v_2 \vdash g \xrightarrow{\rho} v
\]
Representation of ghost state

\[
g \xrightarrow{\rho} \pi \xrightarrow{} \mathbf{v} \quad \triangleq \quad \exists (v_i) \quad \prod_{i \in \mathbb{N}_\rho} g_i \xrightarrow{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v
\]

Two tree shares:

- \(\pi\): permission (what can we do...)
- \(\rho\): location (...to which part)

Composed value:

- \(v\) (the sum is finite)
  (can be any PCM)

\[
\rho_1 \oplus \rho_2 = \rho \quad v_1 \cdot v_2 = v
\]

\[
g \xrightarrow{\rho_1} v_1 \ast g \xrightarrow{\rho_2} v_2 \vdash g \xrightarrow{\rho} v
\]

\[
g \xrightarrow{\rho} \mathbf{v} \vdash \exists \rho_1, \rho_2, v_1, v_2, \quad \rho_1 \oplus \rho_2 = \rho \quad \land \quad v_1 \cdot v_2 = v \quad \land \quad g \xrightarrow{\rho_1} v_1 \ast g \xrightarrow{\rho_2} v_2
\]
Representation of ghost state

\[
g \xrightarrow{\pi_\rho} v \triangleq \exists (v_i) \prod_{i \in \mathbb{N}_\rho} \xrightarrow{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v
\]

Two tree shares:
- \( \pi \): permission (what can we do...)
- \( \rho \): location (...to which part)

Composed value:
- \( v \) (the sum is finite)
  (can be any PCM)

\[
\rho_1 \oplus \rho_2 = \rho \\
v_1 \cdot v_2 = v
\]

\[
g \xrightarrow{\pi_{\rho_1}} v_1 \ast g \xrightarrow{\pi_{\rho_2}} v_2 \vdash g \xrightarrow{\pi_{\rho}} v
\]

\[
g \xrightarrow{\pi_{\rho}} v \vdash \exists \rho_1, \rho_2, \rho_1 \oplus \rho_2 = \rho \land g \xrightarrow{\pi_{\rho_1}} v \ast g \xrightarrow{\pi_{\rho_2}} 1
\]
Representation of ghost state

\[ g \xrightarrow{\rho} v \triangleq \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \xrightarrow{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v \]

Two tree shares:
- \( \pi \): permission (what can we do...)
- \( \rho \): location (...to which part)

Composed value:
- \( v \) (the sum is finite)
  (can be any PCM)

\[ \rho_1 \oplus \rho_2 = \rho \quad v_1 \cdot v_2 = v \]
\[ g \xrightarrow{\rho_1} v_1 \cdot g \xrightarrow{\rho_2} v_2 \vdash g \xrightarrow{\rho} v \]

\[ g \xrightarrow{\rho} v = \exists \rho_1, \rho_2, v_1, v_2, \quad \rho_1 \oplus \rho_2 = \rho \quad \land \quad v_1 \cdot v_2 = v \quad \land \quad g \xrightarrow{\rho_1} v_1 \cdot g \xrightarrow{\rho_2} v_2 \]
Representation of ghost state

\[ g \xrightarrow{\pi} v \triangleq \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \xrightarrow{\pi} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v \]

Two tree shares:
- \( \pi \): permission (what can we do...)
- \( \rho \): location (...to which part)

Composed value:
- \( v \) (the sum is finite)
  (can be any PCM)

\[ g \xrightarrow{\pi} v = \exists \rho_1, \rho_2, v_1, v_2, \quad \rho_1 \oplus \rho_2 = \rho \land v_1 \cdot v_2 = v \land g \xrightarrow{\rho_1} v_1 \ast g \xrightarrow{\rho_2} v_2 \]
Threads sharing memory

$$\{ s \rightarrow R \ast x \rightarrow 0 \}$$
Threads sharing memory

\{ s \to R \ast x \mapsto 0 \}
\{ s \to R \ast x \mapsto 0 \ast g \mapsto 0 \}
Threads sharing memory

\[
\begin{align*}
\{ s \rightarrow^* R \cdot x \rightarrow 0 \} \\
\{ s \rightarrow^* R \cdot x \rightarrow 0 \cdot g \rightarrow 0 \}
\end{align*}
\]

\( V(s); \)
Threads sharing memory

\[
\begin{align*}
\{s \xrightarrow{R \cdot x \mapsto 0} \} \\
\{s \xrightarrow{R \cdot x \mapsto 0 \cdot g \mapsto 0} \}
\end{align*}
\]

\[V(s) ;
\]

\[R \triangleq \exists v \ x \mapsto v \cdot g \mapsto v\]
Threads sharing memory

\{ s \to R \cdot x \mapsto 0 \}
\{ s \to R \cdot x \mapsto 0 \cdot g \mapsto 0 \}

V(s);

\{ s \to R \cdot g \mapsto 0 \}

R \triangleq \exists v \ x \mapsto v \cdot g \mapsto v
Threads sharing memory

\[
\{ s \xrightarrow{\cdot} R \times x \mapsto 0 \}
\]

\[
\{ s \xrightarrow{\cdot} R \times x \mapsto 0 \times g \mapsto 0 \}
\]

\[ V(s); \]

\[
\{ s \xrightarrow{\cdot} R \times g \mapsto 0 \}
\]

\[
\{ s \xrightarrow{\cdot} R \times g \mapsto 0 \times g \mapsto 0 \}
\]

\[ R \triangleq \exists v \ x \mapsto v \times g \mapsto v \]
Threads sharing memory

\[
\begin{align*}
\{s \rightarrow R \ast x \mapsto 0\} \\
\{s \rightarrow R \ast x \mapsto 0 \ast g \mapsto 0\}
\end{align*}
\]

\( V(s); \)

\[
\begin{align*}
\{s \rightarrow R \ast g \mapsto 0\} \\
\{s \rightarrow R \ast g \mapsto 0 \ast g \mapsto 0\}
\end{align*}
\]

\( (P(s); x++; V(s)) \parallel (P(s); x++; V(s)); \)

\[
R \triangleq \exists v \ x \mapsto v \ast g \mapsto v
\]
Threads sharing memory

\{ s \xrightarrow{} R \times x \xrightarrow{} 0 \} \\
\{ s \xrightarrow{} R \times x \xrightarrow{} 0 \times g \xrightarrow{} 0 \}

V(s) ; \\
\{ s \xrightarrow{} R \times g \xrightarrow{} 0 \}

\{ s \xrightarrow{} R \times g \xrightarrow{} 0 \times g \xrightarrow{} 0 \}

(P(s) ; x++ ; V(s)) \parallel (P(s) ; x++ ; V(s)) ; \\
\{ s \xrightarrow{} R \times g \xrightarrow{} 1 \times g \xrightarrow{} 1 \}

R \triangleq \exists v \ x \xrightarrow{} v \times g \xrightarrow{} v
Threads sharing memory

$\{s \xrightarrow{\cdot} R \ast x \mapsto 0\}$
$\{s \xrightarrow{\cdot} R \ast x \mapsto 0 \ast g \mapsto 0\}$

$V(s)$;

$\{s \xrightarrow{\cdot} R \ast g \mapsto 0\}$
$\{s \xrightarrow{\cdot} R \ast g \mapsto 0 \ast g \mapsto 0\}$

$(P(s); x++; V(s)) \parallel (P(s); x++; V(s));$
$\{s \xrightarrow{\cdot} R \ast g \mapsto 1 \ast g \mapsto 1\}$
$\{s \xrightarrow{\cdot} R \ast g \mapsto 2\}$

$R \triangleq \exists v \ x \mapsto v \ast g \mapsto v$
Threads sharing memory

\begin{align*}
\{ s \overset{R * x}{\rightarrow} 0 \} \\
\{ s \overset{R * x}{\rightarrow} 0 * g \overset{0}{\rightarrow} 0 \}
\end{align*}

\begin{align*}
V(s); \\
R \triangleq \exists v \ x \overset{v}{\rightarrow} v * g \overset{v}{\rightarrow} v
\end{align*}

\begin{align*}
\{ s \overset{R * g}{\rightarrow} 0 \} \\
\{ s \overset{R * g}{\rightarrow} 0 * g \overset{0}{\rightarrow} 0 \}
\end{align*}

\begin{align*}
(P(s); x++; V(s)) \parallel (P(s); x++; V(s)); \\
\{ s \overset{R * g}{\rightarrow} 1 * g \overset{1}{\rightarrow} 1 \} \\
\{ s \overset{R * g}{\rightarrow} 2 \}
\end{align*}

P(s);
Threads sharing memory

\[
\begin{align*}
\{ s \overset{R \ast x}{\rightarrow} 0 \} \\
\{ s \overset{R \ast x}{\rightarrow} 0 \ast g \overset{0}{\rightarrow} 0 \}
\end{align*}
\]

\[
V(s); \quad R \triangleq \exists v \ x \overset{v \ast g}{\rightarrow} v
\]

\[
\begin{align*}
\{ s \overset{R \ast g}{\rightarrow} 0 \} \\
\{ s \overset{R \ast g}{\rightarrow} 0 \ast g \overset{0}{\rightarrow} 0 \}
\end{align*}
\]

\[
(P(s); \ x++; \ V(s)) \ || \ (P(s); \ x++; \ V(s));
\]

\[
\begin{align*}
\{ s \overset{R \ast g}{\rightarrow} 1 \ast g \overset{1}{\rightarrow} 1 \} \\
\{ s \overset{R \ast g}{\rightarrow} 2 \}
\end{align*}
\]

\[
P(s); \\
\{ s \overset{R \ast x}{\rightarrow} 2 \ast g \overset{2}{\rightarrow} 2 \}
\]
Threads sharing memory

\[
\{ s \xrightarrow{\square} R \times x \rightarrow 0 \} \\
\{ s \xrightarrow{\square} R \times x \rightarrow 0 \times g \rightarrow 0 \}
\]

\[
V(s) ;
\]

\[
\{ s \xrightarrow{\square} R \times g \rightarrow 0 \} \\
\{ s \xrightarrow{\square} R \times g \rightarrow 0 \times g \rightarrow 0 \}
\]

\[
(P(s) ; x++ ; V(s)) \parallel (P(s) ; x++ ; V(s));
\]

\[
\{ s \xrightarrow{\square} R \times g \rightarrow 1 \times g \rightarrow 1 \} \\
\{ s \xrightarrow{\square} R \times g \rightarrow 2 \}
\]

\[
P(s) ;
\]

\[
\{ s \xrightarrow{\square} R \times x \rightarrow 2 \times g \rightarrow 2 \} \\
\{ s \xrightarrow{\square} R \times x \rightarrow 2 \}
\]

\[
R \triangleq \exists v \ x \rightarrow v \times g \rightarrow v
\]
Threads sharing memory

\[
\{ s \rightarrow R \cdot x \mapsto 0 \} \\
\{ s \rightarrow R \cdot x \mapsto 0 \cdot g \mapsto 0 \}
\]

\[ V(s); \quad R \triangleq \exists v \ x \mapsto v \cdot g \mapsto v \]

\[
\{ s \rightarrow R \cdot g \mapsto 0 \} \\
\{ s \rightarrow R \cdot g \mapsto 0 \cdot g \mapsto 0 \}
\]

\[ (P(s); \ x++; \ V(s)) \parallel (P(s); \ x++; \ V(s)); \]

\[
\{ s \rightarrow R \cdot g \mapsto 1 \cdot g \mapsto 1 \} \\
\{ s \rightarrow R \cdot g \mapsto 2 \}
\]

\[ P(s); \]

\[
\{ s \rightarrow R \cdot x \mapsto 2 \cdot g \mapsto 2 \} \\
\{ s \rightarrow R \cdot x \mapsto 2 \}
\]

assert(x == 2);
Threads sharing memory

\[
\{ s \rightarrow R \ast x \mapsto 0 \} \\
\{ s \rightarrow R \ast x \mapsto 0 \ast g \mapsto 0 \} \\
V(s); \\
\{ s \rightarrow R \ast g \mapsto 0 \} \\
\{ s \rightarrow R \ast 0 \ast g \mapsto 0 \} \\
(P(s); x++; V(s)) \parallel (P(s); x++; V(s)); \\
\{ s \rightarrow R \ast g \mapsto 1 \ast g \mapsto 1 \} \\
\{ s \rightarrow R \ast g \mapsto 2 \} \\
P(s); \\
\{ s \rightarrow R \ast x \mapsto 2 \ast g \mapsto 2 \} \\
\{ s \rightarrow R \ast x \mapsto 2 \} \\
assert(x == 2); \\
\{ s \rightarrow R \ast x \mapsto 2 \} \\
\]

\[ R \triangleq \exists v \ x \mapsto v \ast g \mapsto v \]
Threads sharing memory

\{s \xrightarrow{R \ast x} \bullet \mapsto 0\}
\{s \xrightarrow{R \ast x} \mapsto 0 \ast g \mapsto 0\}

V(s);  \hspace{0.5cm} R \triangleq \exists v \ x \mapsto v \ast g \mapsto v

\{s \xrightarrow{R \ast g} \bullet \mapsto 0\}
\{s \xrightarrow{R \ast g} \mapsto 0 \ast g \mapsto 0\}

(P(s); x++; V(s)) || (P(s); x++; V(s));

\{s \xrightarrow{R \ast g} 1 \ast g \mapsto 1\}
\{s \xrightarrow{R \ast g} 2\}

P(s);
\{s \xrightarrow{R \ast x} 2 \ast g \mapsto 2\}
\{s \xrightarrow{R \ast x} 2\}

assert(x == 2);
\{s \xrightarrow{R \ast x} 2\}  \hspace{1cm} \text{safe!}
... we catch back on Nanevski’s subjective views

Thread 1

\[ x = 0; \]

\[ \text{shared resource} \]

Thread 2

\[ x + 1; \]

\[ \text{verified: } x = 2 \]

\[ \text{Safe!} \]

... we catch back on Nanevski’s subjective views

Thread 1                      Shared resource                      Thread 2

x = 0;
V(s);

x++;
V(s);

x++;
V(s);

assert(x == 2);

Safe!
... we catch back on Nanevski’s subjective views

Thread 1

\[
x = 0;  
V(s);  
x++;  
V(s);  
x++;  
V(s);  
\]

Shared resource

Thread 2

\[
P(s);  
\]

Safe!
... we catch back on Nanevski’s subjective views

Thread 1

\[
x = 0; \\
V(s); \\
x++;
\]

Shared resource

Thread 2

\[
P(s); \\
x++; \\
V(s); \\
x++;
\]
... we catch back on Nanevski’s subjective views

Thread 1  

\[
x = 0;  
\]

\[
V(s);  
\]

Shared resource

Thread 2

\[x\]

\[
P(s);  
\]

\[
x++;  
\]

\[
V(s);  
\]
... we catch back on Nanevski’s subjective views

Thread 1

\[ x = 0; \]
\[ V(s); \]
\[ P(s); \]
\[ x++; \]
\[ V(s); \]

Thread 2

\[ P(s); \]
\[ x++; \]
\[ V(s); \]

Shared resource

\[ x \]
\[ 0 \]
... we catch back on Nanevski’s subjective views

Thread 1

\[
x = 0;
V(s);
\]

\[
P(s);
x++;
\]

Thread 2

\[
V(s);
P(s);
x++;
V(s);
\]

\[
\text{Safe!}
\]
... we catch back on Nanevski’s subjective views

Thread 1

\[ x = 0; \]
\[ V(s); \]
\[ P(s); \]
\[ x++; \]
\[ V(s); \]

Shared resource

\[ x \]

Thread 2

\[ P(s); \]
\[ x++; \]
\[ V(s); \]
... we catch back on Nanevski’s subjective views

Thread 1

\[ x = 0; \]
\[ V(s); \]
\[ P(s); \]
\[ x++; \]
\[ V(s); \]

Thread 2

\[ P(s); \]
\[ x++; \]
\[ V(s); \]

Shared resource

\[
\begin{array}{cc}
0 & \oplus \\
\hline
1 & 1
\end{array}
\]
... we catch back on Nanevski’s subjective views

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Shared resource</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V(s);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P(s);</td>
<td></td>
<td>P(s);</td>
</tr>
<tr>
<td>x++;</td>
<td></td>
<td>x++;</td>
</tr>
<tr>
<td>V(s);</td>
<td></td>
<td>V(s);</td>
</tr>
<tr>
<td>P(s);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
... we catch back on Nanevski’s subjective views

Thread 1

\[ x = 0; \]
\[ V(s); \]
\[ P(s); \]
\[ x++; \]
\[ V(s); \]

\[ P(s); \]
\[ assert(x == 2); \]

Thread 2

\[ P(s); \]
\[ x++; \]
\[ V(s); \]

Shared resource

\[ x \]
\[ 2 \]
... we catch back on Nanevski’s subjective views

Thread 1

\[ \begin{align*} x &= 0; \\ V(s); \end{align*} \]

Thread 2

\[ \begin{align*} x &= \text{shared resource} \\ V(s); \\ P(s); \\ x &= x + 1; \\ V(s); \end{align*} \]

\[ \text{assert}(x == 2); \quad \text{Safe!} \]
#include <pthread.h>
#include <semaphore.h>

void assert(int i) { i = 1/i; }

sem_t s;
int x;

void* f(void *arg) {
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_exit(NULL);
}

int main (void) {
    pthread_t th;
    x = 0;
    sem_init(&s, 0, 0);
    sem_post(&s);
    pthread_create(&th, NULL, f, (void*)&x);
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_join(th, NULL);
    sem_wait(&s);
    sem_destroy(&s);
    assert(x >= 0);
    return 0;
}
Summary

1 sem_wait grants access

```c
#include <pthread.h>
#include <semaphore.h>

void assert(int i) {i = 1/i;}

sem_t s;
int x;

void* f(void *arg) {
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_exit(NULL);
}

int main (void) {
    pthread_t th;
    x = 0;
    sem_init(&s, 0, 0);
    sem_post(&s);
    pthread_create(&th, NULL, f, (void**)&x);
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_join(th, NULL);
    sem_wait(&s);
    sem_destroy(&s);
    assert(x >= 0);
    return 0;
}
Summary

1. `sem_wait` grants access
2. `sem_post` gives it away

```c
#include <pthread.h>
#include <semaphore.h>

void assert(int i) { i = 1/i; }

sem_t s;
int x;

void* f(void *arg) {
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_exit(NULL);
}

int main (void) {
    pthread_t th;
    x = 0;
    sem_init(&s, 0, 0);
    sem_post(&s);
    pthread_create(&th, NULL, f, (void*)&x);
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_join(th, NULL);
    sem_wait(&s);
    sem_destroy(&s);
    assert(x >= 0);
    return 0;
}
```
Summary

1. `sem_wait` grants access
2. `sem_post` gives it away
3. We want knowledge about x’s value

```c
#include <pthread.h>
#include <semaphore.h>

void assert(int i) { i = 1/i; }

sem_t s;
int x;

void* f(void *arg) {
  sem_wait(&s);
  x++;
  sem_post(&s);
pthread_exit(NULL);
}

int main (void) {
  pthread_t th;
  x = 0;
  sem_init(&s, 0, 0);
  sem_post(&s);
pthread_create(&th, NULL, f, (void*)&x);
  sem_wait(&s);
  x++;
  sem_post(&s);
pthread_join(th, NULL);
  sem_wait(&s);
  sem_destroy(&s);
  assert(x >= 0);
  return 0;
}
```
Summary

1. `sem_wait` grants access
2. `sem_post` gives it away
3. We want knowledge about `x`'s value

Concurrent Separation Logic (O'Hearn'04)

```c
#include <pthread.h>
#include <semaphore.h>

void assert(int i) { i = 1/i; }

sem_t s;
int x;

void* f(void *arg) {
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_exit(NULL);
}

int main (void) {
    pthread_t th;
    x = 0;
    sem_init(&s, 0, 0);
    sem_post(&s);
    pthread_create(&th, NULL, f, (void*)&x);
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_join(th, NULL);
    sem_wait(&s);
    sem_destroy(&s);
    assert(x >= 0);
    return 0;
}
```
Summary

1. `sem_wait` grants access
2. `sem_post` gives it away
3. We want knowledge about $x$’s value
   Concurrent Separation Logic (O’Hearn’04)
4. We can create locks and spawn threads
   CSL with first-class locks and threads
   (Gotsman’07, Hobor’08)

```c
#include <pthread.h>
#include <semaphore.h>

void assert(int i) { i = 1/i; }

sem_t s;
int x;

void* f(void *arg) {
  sem_wait(&s);
  x++;
  sem_post(&s);
  pthread_exit(NULL);
}

int main (void) {
  pthread_t th;
  x = 0;
  sem_init(&s, 0, 0);
  sem_post(&s);
  pthread_create(&th, NULL, f, (void*) &x);
  sem_wait(&s);
  x++;
  sem_post(&s);
  pthread_join(th, NULL);
  sem_wait(&s);
  sem_destroy(&s);
  assert(x >= 0);
  return 0;
}
```
#include <pthread.h>
#include <semaphore.h>

void assert(int i) { i = 1/i; }

sem_t s;
int x;

void* f(void *arg) {
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_exit(NULL);
}

int main (void) {
    pthread_t th;
    x = 0;
    sem_init(&s, 0, 0);
    sem_post(&s);
    pthread_create(&th, NULL, f, (void*)&x);
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_join(th, NULL);
    sem_wait(&s);
    sem_destroy(&s);
    assert(x >= 0);
    return 0;
}

## Summary

1. `sem_wait` grants access
2. `sem_post` gives it away
3. We want knowledge about x’s value
   - Concurrent Separation Logic (O’Hearn’04)
4. We can create locks and spawn threads
   - CSL with first-class locks and threads
     - Gotsman’07, Hobor’08
5. We need to join threads and transfer back the permissions from f to main.
Summary

1. sem_wait grants access
2. sem_post gives it away
3. we want knowledge about x’s value
   Concurrent Separation Logic (O’Hearn’04)
4. we can create locks and spawn threads
   CSL with first-class locks and threads
   (Gotsman’07, Hobor’08)
5. we need to join threads and transfer back
   the permissions from f to main.
6. maintaining x >= 0 is one thing, but how
   to ensure x == 2 in the end?

```c
#include <pthread.h>
#include <semaphore.h>

void assert(int i) {i = 1/i;}

sem_t s;
int x;

void* f(void *arg) {
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_exit(NULL);
}

int main (void) {
    pthread_t th;
    x = 0;
    sem_init(&s, 0, 0);
    sem_post(&s);
    pthread_create(&th, NULL, f, (void*)&x);
    sem_wait(&s);
    x++;
    sem_post(&s);
    pthread_join(th, NULL);
    sem_wait(&s);
    sem_destroy(&s);
    assert(x == 0);
    return 0;
}
```
#include <pthread.h>
#include <semaphore.h>

void assert(int i) { i = 1/i; }

sem_t s;
int x;

void* f(void *arg) {
  sem_wait(&s);
  x++;
  sem_post(&s);
  pthread_exit(NULL);
}

int main (void) {
  pthread_t th;
  x = 0;
  sem_init(&s, 0, 0);
  sem_post(&s);
  pthread_create(&th, NULL, f, (void*) &x);
  sem_wait(&s);
  x++;
  sem_post(&s);
  pthread_join(th, NULL);
  sem_wait(&s);
  sem_destroy(&s);
  assert(x >= 0);
  return 0;
}

Summary

1. sem_wait grants access
2. sem_post gives it away
3. we want knowledge about x’s value
   Concurrent Separation Logic (O’Hearn’04)
4. we can create locks and spawn threads
   CSL with first-class locks and threads
   (Gotsman’07, Hobor’08)
5. we need to join threads and transfer back
   the permissions from f to main.
6. maintaining x >= 0 is one thing, but how
   to ensure x == 2 in the end?
   Ghost variables (∼folklore)
Summary

1. `sem_wait` grants access
2. `sem_post` gives it away
3. We want knowledge about x’s value
   Concurrent Separation Logic (O’Hearn’04)
4. We can create locks and spawn threads
   CSL with first-class locks and threads
   (Gotsman’07, Hobor’08)
5. We need to join threads and transfer back the permissions from f to main.
6. Maintaining x >= 0 is one thing, but how to ensure x == 2 in the end?
   Ghost variables (∼folklore)
   Subjective CSL (Nanevski’14), Iris (JSS+’15)
### Summary

1. `sem_wait` grants access
2. `sem_post` gives it away
3. We want knowledge about $x$'s value
   
   Concurrent Separation Logic (O’Hearn’04)
4. We can create locks and spawn threads
   
   CSL with first-class locks and threads
   (Gotsman’07, Hobor’08)
5. We need to join threads and transfer back the permissions from $f$ to main.
6. Maintaining $x \geq 0$ is one thing, but how to ensure $x == 2$ in the end?

   Ghost variables (∼folklore)

Subjective CSL (Nanevski’14), Iris (JSS+’15)

$\simeq$ our ghost state
Program patterns

We can do:

- simple $x++ | x++$ variants,
- (some) producer/consumer implementations,
- distributed initialize-once.

We expect:

- parallel sorting algorithms,
- other producer/consumer implementations,
- parallel tree/graph traversals,
- ... waiting from our sponsor to provide program patterns.

We don't do:

- RCU,
- races, low-level barriers,
- lock-free implementations.
Program patterns

We can do:

- simple $x++ \mid x++$ variants,
- (some) producer/consumer implementations,
- distributed initialize-once.

We expect:

- parallel sorting algorithms,
- other producer/consumer implementations,
- parallel tree/graph traversals,
- ... waiting from our sponsor to provide program patterns.
Program patterns

We can do:

- simple $x++ | x++$ variants,
- (some) producer/consumer implementations,
- distributed initialize-once.

We expect:

- parallel sorting algorithms,
- other producer/consumer implementations,
- parallel tree/graph traversals,
- ... waiting from our sponsor to provide program patterns.

We don’t do:

- RCU,
- races, low-level barriers,
- lock-free implementations.
Thank you for having me!

\[ g \xrightarrow{\pi} v \xrightarrow{\rho} \]

https://github.com/PrincetonUniversity/VST/tree/concurrency