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 sofware development

Verified software development:

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

Top to bottom verified sofware 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.

Top to bottom verified sofware 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 sofware development

Verified software development:

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

Existing tools perform well:

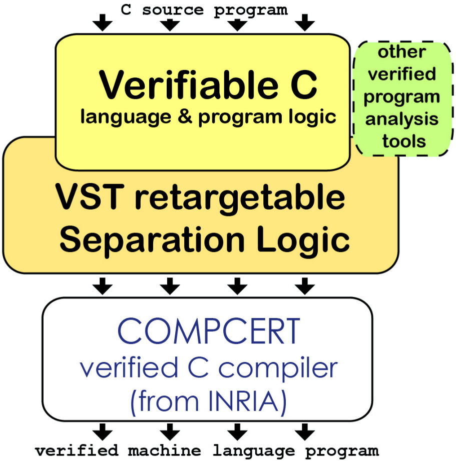
- reasoning: powerful program logics and analysers,
- translations: CompCert certified compiler,
- models of weak memory for different architectures.

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.

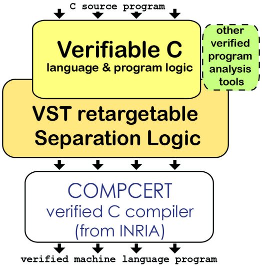
Architecture of VST



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

$P ::= \ \ldots$	$v \Downarrow 4$	local variables
	$p \mapsto 4, p \mapsto \_$	pointers, shape
	$f : P \to Q$	function pointers (indirection)
	$!!P$	embedding of Coq propositions
	$\wedge, \vee$	usual logical operators
	$\mu x.P$	recursion
	$\forall x.P, \exists x.P$	impredicative quantification
	$P * Q, P \twoheadrightarrow Q$	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 **verifiable C**

```c
#include <stddef.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 *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)->tail);`)

(the transformation could be done automatically, but we still need to reason on this program)

Proof of merge.c

File Edit Options Buffers Tools Coq Proof-General Holes Help

```
  extract_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 a0] b0] merged0] a_0] b_0] c_0] begin0].
+ (* Loop: precondition => invariant *)
 Exists cond a b merged a_ b_ c_ begin; entailer!.
+ (* Loop: condition has nice format *)
now entailer!.
+ (* Loop body preserves invariant *)
clear - SH HRE H1 H2.
rename cond0 into cond, a0 into a, b0 into b, merged0 into merged,
 a_0 into a_, b_0 into b_, c_0 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 *)
  normalize. now intuition.
normalize.
intros a_'.
normalize.
(* Now the command [va = a->head] can proceed *)
rewrite list_cell_field_at.
forward.

rewrite lseg_unfold with (v1:=b_).
destruct b as [|vb b']; simpl.
  (* [b] cannot be empty *)
  normalize; now intuition.
normalize.
intros b_'.
normalize.
clear H2 H3.
```

`-:---  verif_merge.v    33% (234,36)  Git-concurrency  (`

```
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 tlist ret_; temp _cond (Vint cond))
    SEP  (`(lseg LS sh (map Vint a) a_ nullval);
    `(lseg LS sh (map Vint b) b_ nullval);
    `(data_at Tsh tlist (if merged then Vundef else begin) ret_);
    `(lseg LS sh (map Vint (butlast merged)) begin c_);
    `(if merged
      then emp
      else data_at sh t_struct_list (Vint (last merged), Vundef) c_)))
   (Ssequence
      (Sset _va
         (Efield
            (Ederef (Etempvar _a (tptr (Tstruct _list noattr)))
               (Tstruct _list noattr)) _head tint)) MORE_COMMANDS)
   POSTCONDITION
```

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

`U:%%-  *response*   All (1,0)     (Coq Response vl`

Concurrent programs

Simple concurrent program

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

Proof of simple concurrent program

$$\{x \mapsto \_ * y \mapsto \_\}$$

```
x = 0;
```

$$\{x \mapsto 0 * y \mapsto \_\}$$

```
y = 0;
```

$$\{x \mapsto 0 * y \mapsto 0\}$$

```
x++ || y++;
```

$$(*) \quad \{x \mapsto 1 * y \mapsto 1\}$$

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

$$\{x \mapsto 1 * y \mapsto 1\}$$

Proof of simple concurrent program

$$\{x \mapsto \_ * y \mapsto \_\}$$
```
x = 0;
```
$$\{x \mapsto 0 * y \mapsto \_\}$$
```
y = 0;
```
$$\{x \mapsto 0 * y \mapsto 0\}$$
```
x++ || y++;
```
$$(*) \quad \{x \mapsto 1 * y \mapsto 1\}$$
```
assert(x + y == 2);
```
$$\{x \mapsto 1 * y \mapsto 1\}$$

$$(*) \ \frac{\{x \mapsto 0\}\ \texttt{x++}\ \{x \mapsto 1\} \qquad \{y \mapsto 0\}\ \texttt{y++}\ \{y \mapsto 1\}}{\{x \mapsto 0 * y \mapsto 0\}\ \texttt{x++ || y++}\ \{x \mapsto 1 * y \mapsto 1\}}$$

(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;

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;

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.

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:

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

$$\pi \ ::= \ \texttt{Freeable} \mid \texttt{Writable} \mid \texttt{Readable} \mid \texttt{Nonempty}$$

Permissions in CompCert

CompCert memory model, version 2:

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

$$\pi \ ::= \ \texttt{Freeable} > \texttt{Writable} > \texttt{Readable} > \texttt{Nonempty}$$

Permissions in CompCert

CompCert memory model, version 2:

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

$$\pi \ ::= \ \texttt{Freeable} > \texttt{Writable} > \texttt{Readable} > \texttt{Nonempty}$$

- 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:

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

$$\pi \;::=\; \texttt{Freeable} > \texttt{Writable} > \texttt{Readable} > \texttt{Nonempty}$$

- 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

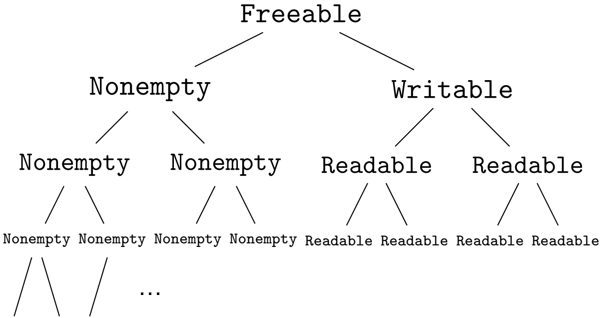
Refinement of CompCert's permissions:

$$\pi \quad ::= \quad \bullet \quad \mid \quad \circ \quad \mid \quad \overset{\wedge}{\pi_1 \ \ \pi_2}$$

Joining permissions:

$$\overset{\wedge}{\bullet \ \ \circ} \oplus \overset{\wedge}{\circ \ \ \bullet} = \overset{\wedge}{\bullet \ \ \bullet} = \bullet \qquad \frac{\pi = \pi_1 \oplus \pi_2}{p \overset{\pi}{\mapsto} v = p \overset{\pi_1}{\mapsto} v * p \overset{\pi_2}{\mapsto} v}$$

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

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

no race!

Threads sharing memory, using semaphores in Pthreads

```c
#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;
}
```

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

Threads sharing memory, using binary semaphores

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

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

Threads sharing memory, using binary semaphores

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

$$
\begin{array}{l||l}
\{s \mapsto \mathsf{lock}[x]\} & \{s \mapsto \mathsf{lock}[x]\} \\
\texttt{P(s);} & \texttt{P(s);} \\
\{s \mapsto \mathsf{lock}[x] * x \overset{\bullet}{\mapsto} \_\} & \{s \mapsto \mathsf{lock}[x] * x \overset{\bullet}{\mapsto} \_\} \\
\texttt{x++;} & \texttt{x++;} \\
\{s \mapsto \mathsf{lock}[x] * x \overset{\bullet}{\mapsto} \_\} & \{s \mapsto \mathsf{lock}[x] * x \overset{\bullet}{\mapsto} \_\} \\
\texttt{V(s);} & \texttt{V(s);} \\
\{s \mapsto \mathsf{lock}[x]\} & \{s \mapsto \mathsf{lock}[x]\}
\end{array}
$$

Threads sharing memory

Permissions can be refined to *lock invariants*:

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$
`P(s);`	`P(s);`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$
`x++;`	`x++;`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$
`V(s);`	`V(s);`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$

Threads sharing memory

Atomicity is not mandatory:

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$
`P(s);`	`P(s);`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$
`a = x;`	`a = x;`
`x = a - 2; // x can be negative here`	`x = a - 3;`
`x = a + 1;`	`x = a + 1;`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$
`V(s);`	`V(s);`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$

Threads sharing memory

Coming back to `x++`:

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$
`P(s);`	`P(s);`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$
`x++;`	`x++;`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \overset{\bullet}{\mapsto} n\}$
`V(s);`	`V(s);`
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$	$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$

Threads sharing memory

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \mapsto n] * x \stackrel{\bullet}{\mapsto} \_\}$

`x = 0;`

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \stackrel{\bullet}{\mapsto} n] * x \stackrel{\bullet}{\mapsto} 0\}$

`V(s);`

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \stackrel{\bullet}{\mapsto} n]\}$

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

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \stackrel{\bullet}{\mapsto} n]\}$

`P(s);`

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \stackrel{\bullet}{\mapsto} n] * \exists n \geq 0\ x \mapsto n\}$

`assert(x >= 0);`

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \stackrel{\bullet}{\mapsto} n] * \exists n \geq 0\ x \mapsto n\}$

Threads sharing memory

$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \mapsto n] * x \overset{\bullet}{\mapsto} \_\}$
```
x = 0;
```
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * x \overset{\bullet}{\mapsto} 0\}$
```
V(s);
```
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$
```
(P(s); x++; V(s)) || (P(s); x++; V(s));
```
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n]\}$
```
P(s);
```
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \mapsto n\}$
```
assert(x >= 0);
```
$\{s \mapsto \mathsf{lock}[\exists n \geq 0\ x \overset{\bullet}{\mapsto} n] * \exists n \geq 0\ x \mapsto n\}$

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.

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

```
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; x1 = 0; x2 = 0;
V(s);
(P(s); x1++; x++; V(s)) || (P(s); x2++; x++; V(s));
P(s);
assert(x == 2);
```

Ghost variables

A new invariant relying on ghost variables: $R = \exists n_1, n_2 \begin{array}{l} x \overset{\bullet}{\mapsto} n_1 + n_2 \\ * \; x_1 \overset{\LEFTcircle}{\mapsto} n_1 \\ * \; x_2 \overset{\LEFTcircle}{\mapsto} n_2 \end{array}$

$\{l \boxdot\!\!\overset{\LEFTcircle}{\mapsto} R * x_2 \overset{\LEFTcircle}{\mapsto} 0\}$

```
P(l);
```

$\{l \boxdot\!\!\overset{\LEFTcircle}{\mapsto} R * \exists n_1 \; x_2 \overset{\bullet}{\mapsto} 0 * x \overset{\bullet}{\mapsto} n_1 + 0 * x_1 \overset{\LEFTcircle}{\mapsto} n_1\}$

$\{l \boxdot\!\!\overset{\LEFTcircle}{\mapsto} R * x_2 \overset{\bullet}{\mapsto} 0 * x \overset{\bullet}{\mapsto} n_1 + 0 * x_1 \overset{\LEFTcircle}{\mapsto} n_1\}$

```
x2++;
```

$\{l \boxdot\!\!\overset{\LEFTcircle}{\mapsto} R * x_2 \overset{\bullet}{\mapsto} 1 * x \overset{\bullet}{\mapsto} n_1 + 0 * x_1 \overset{\LEFTcircle}{\mapsto} n_1\}$

```
x++;
```

$\{l \boxdot\!\!\overset{\LEFTcircle}{\mapsto} R * x_2 \overset{\bullet}{\mapsto} 1 * x \overset{\bullet}{\mapsto} n_1 + 1 * x_1 \overset{\LEFTcircle}{\mapsto} n_1\}$

$\{l \boxdot\!\!\overset{\LEFTcircle}{\mapsto} R * x_2 \overset{\LEFTcircle}{\mapsto} 1 * R\}$

```
V(l);
```

$\{l \boxdot\!\!\overset{\LEFTcircle}{\mapsto} R * x_2 \overset{\LEFTcircle}{\mapsto} 1\}$

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} \_\}$

`x = 0; x1 = 0; x2 = 0;`

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * x_1 \overset{\bullet}{\mapsto} 0 * x_2 \overset{\bullet}{\mapsto} 0\}$

`V(s);`

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x_1 \overset{\circ}{\mapsto} 0 * x_2 \overset{\circ}{\mapsto} 0\}$

`(P(s); x1++; x++; V(s)) || (P(s); x2++; x++; V(s));`

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x_1 \overset{\circ}{\mapsto} 1 * x_2 \overset{\circ}{\mapsto} 1\}$

`P(s);`

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2 * x_1 \overset{\bullet}{\mapsto} 1 * x_2 \overset{\bullet}{\mapsto} 1\}$

`assert(x == 2);`

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2 * x_1 \overset{\bullet}{\mapsto} 1 * x_2 \overset{\bullet}{\mapsto} 1\}$

Threads sharing memory

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} \_\}$

`x = 0; x1 = 0; x2 = 0;`

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} 0 * x_1 \overset{\bullet}{\mapsto} 0 * x_2 \overset{\bullet}{\mapsto} 0\}$

`V(s);`

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x_1 \overset{\circ}{\mapsto} 0 * x_2 \overset{\circ}{\mapsto} 0\}$

`(P(s); x1++; x++; V(s)) || (P(s); x2++; x++; V(s));`

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x_1 \overset{\circ}{\mapsto} 1 * x_2 \overset{\circ}{\mapsto} 1\}$

`P(s);`

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} 2 * x_1 \overset{\bullet}{\mapsto} 1 * x_2 \overset{\bullet}{\mapsto} 1\}$

`assert(x == 2);`

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} 2 * x_1 \overset{\bullet}{\mapsto} 1 * x_2 \overset{\bullet}{\mapsto} 1\}$

The above program is safe.

But

Problems:

- unbounded number of ghost variables?

But

Problems:

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

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?

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 $f : \{P\} \to \{Q\}$):

$$\frac{\{\exists g\ g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}} \qquad \frac{\{g \overset{\bullet}{\mapsto} v' * P\}\ c\ \{Q\}}{\{g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}$$

$$g \overset{\bullet}{\mapsto} v \ = \ g \overset{◐}{\mapsto} v \ * \ g \overset{◑}{\mapsto} v$$

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 $f : \{P\} \to \{Q\}$):

$$\frac{\{\exists g\ g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}} \qquad \frac{\{g \overset{\bullet}{\mapsto} v' * P\}\ c\ \{Q\}}{\{g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}$$

$$g \overset{\bullet}{\mapsto} v \ = \ g \overset{\LEFTcircle}{\mapsto} v \ * \ g \overset{\RIGHTcircle}{\mapsto} v$$

- Importantly, when we own $g \overset{\LEFTcircle}{\mapsto} v$ or $\exists v\ g \overset{\LEFTcircle}{\mapsto} v$ 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?
- ... isn't it a *logical* problem?

Solution:

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

$$\frac{\{\exists g\ g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}} \qquad \frac{\{g \overset{\bullet}{\mapsto} v' * P\}\ c\ \{Q\}}{\{g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}$$

$$g \overset{\bullet}{\mapsto} v \ = \ g \overset{\LEFTcircle}{\mapsto} v \ * \ g \overset{\RIGHTcircle}{\mapsto} v$$

- Importantly, when we own $g \overset{\LEFTcircle}{\mapsto} v$ or $\exists v\ g \overset{\LEFTcircle}{\mapsto} v$ 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 $f : \{P\} \to \{Q\}$):

$$\frac{\{\exists g\ g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}} \qquad \frac{\{g \overset{\bullet}{\mapsto} v' * P\}\ c\ \{Q\}}{\{g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}$$

$$g \overset{\bullet}{\mapsto} v \ = \ g \overset{\LEFTcircle}{\mapsto} v \ * \ g \overset{\RIGHTcircle}{\mapsto} v$$

- Importantly, when we own $g \overset{\LEFTcircle}{\mapsto} v$ or $\exists v\ g \overset{\LEFTcircle}{\mapsto} v$ 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 $f : \{P\} \to \{Q\}$):

$$\frac{\{\exists g\ g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}} \qquad \frac{\{g \overset{\bullet}{\mapsto} v' * P\}\ c\ \{Q\}}{\{g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}$$

$$g \overset{\bullet}{\mapsto} v \ = \ g \overset{\LEFTcircle}{\mapsto} v \ * \ g \overset{\RIGHTcircle}{\mapsto} v$$

- Importantly, when we own $g \overset{\LEFTcircle}{\mapsto} v$ or $\exists v\ g \overset{\LEFTcircle}{\mapsto} v$ 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 $f : \{P\} \to \{Q\}$):

$$\frac{\{\exists g\ g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}} \qquad \frac{\{g \overset{\bullet}{\mapsto} v' * P\}\ c\ \{Q\}}{\{g \overset{\bullet}{\mapsto} v * P\}\ c\ \{Q\}}$$

$$g \overset{\bullet}{\mapsto} v \ = \ g \overset{◐}{\mapsto} v \ * \ g \overset{◑}{\mapsto} v$$

- Importantly, when we own $g \overset{◐}{\mapsto} v$ or $\exists v\ g \overset{◐}{\mapsto} v$ 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 $\mathbb{N}$:

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

and more that once:

$$\mathbb{N} = \biguplus_{k \in \mathbb{N}} 2^k(1 + 2\mathbb{N}) - 1$$

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!

$$t ::= \bullet \mid \circ \mid \genfrac{}{}{0pt}{}{\wedge}{t_1 \quad t_2} \qquad / \qquad \genfrac{}{}{0pt}{}{\wedge}{\bullet \quad \bullet} \equiv \bullet, \ \genfrac{}{}{0pt}{}{\wedge}{\circ \quad \circ} \equiv \circ$$

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!

$$t ::= \bullet \mid \circ \mid \overset{\wedge}{t_1 \quad t_2} \qquad / \qquad \overset{\wedge}{\bullet \quad \bullet} \equiv \bullet, \overset{\wedge}{\circ \quad \circ} \equiv \circ$$

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

$$\mathbb{N}_\bullet = \mathbb{N} \qquad \mathbb{N}_\circ = \emptyset \qquad \mathbb{N}_{\left(\overset{\wedge}{t_1 \quad t_2}\right)} = 2\mathbb{N}_{t_1} \uplus (1 + 2\mathbb{N}_{t_2})$$

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!

$$t ::= \bullet \mid \circ \mid \overset{\wedge}{t_1 \quad t_2} \qquad / \qquad \overset{\wedge}{\bullet \quad \bullet} \equiv \bullet, \; \overset{\wedge}{\circ \quad \circ} \equiv \circ$$

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

$$\mathbb{N}_\bullet = \mathbb{N} \qquad \mathbb{N}_\circ = \emptyset \qquad \mathbb{N}_{\left(\overset{\wedge}{t_1 \quad t_2}\right)} = 2\mathbb{N}_{t_1} \uplus (1 + 2\mathbb{N}_{t_2})$$

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

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

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

We have encountered this problem before!

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

$$t ::= \bullet \mid \circ \mid \overset{\wedge}{t_1 \quad t_2} \qquad / \qquad \overset{\wedge}{\bullet \quad \bullet} \equiv \bullet, \ \overset{\wedge}{\circ \quad \circ} \equiv \circ$$

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

$$\mathbb{N}_\bullet = \mathbb{N} \qquad \mathbb{N}_\circ = \emptyset \qquad \mathbb{N}_{\left(\overset{\wedge}{t_1 \quad t_2}\right)} = 2\mathbb{N}_{t_1} \uplus (1 + 2\mathbb{N}_{t_2})$$

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

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

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

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

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \quad \triangleq \quad \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \wedge \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \quad \triangleq \quad \exists (v_i) \;\; \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \wedge \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Two tree shares:

- π: permission (what can we do...)
- ρ: location (...to which part)

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \quad \triangleq \; \exists (v_i) \; \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \wedge \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Two tree shares:

- π: permission (what can we do...)
- ρ: location (...to which part)

Composed value:

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

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \quad \triangleq \quad \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \wedge \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Two tree shares:

- π: permission (what can we do...)
- ρ: location (...to which part)

Composed value:

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

$$\frac{\pi_1 \oplus \pi_2 = \pi}{g \overset{\pi_1}{\underset{\rho}{\mapsto}} v * g \overset{\pi_2}{\underset{\rho}{\mapsto}} v = g \overset{\pi}{\underset{\rho}{\mapsto}} v}$$

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \quad \triangleq \quad \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \wedge \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Two tree shares:

- π: permission (what can we do...)
- ρ: location (...to which part)

Composed value:

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

$$\frac{\rho_1 \oplus \rho_2 = \rho \qquad v_1 \cdot v_2 = v}{g \overset{\pi}{\underset{\rho_1}{\mapsto}} v_1 * g \overset{\pi}{\underset{\rho_2}{\mapsto}} v_2 \vdash g \overset{\pi}{\underset{\rho}{\mapsto}} v}$$

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \quad \triangleq \quad \exists (v_i) \ \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \wedge \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Two tree shares:

- π: permission (what can we do...)
- ρ: location (...to which part)

Composed value:

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

$$\frac{\rho_1 \oplus \rho_2 = \rho \qquad v_1 \cdot v_2 = v}{g \overset{\pi}{\underset{\rho_1}{\mapsto}} v_1 * g \overset{\pi}{\underset{\rho_2}{\mapsto}} v_2 \vdash g \overset{\pi}{\underset{\rho}{\mapsto}} v}$$

$$\frac{}{g \overset{\pi}{\underset{\rho}{\mapsto}} v \vdash \exists \rho_1, \rho_2, v_1, v_2, \quad \rho_1 \oplus \rho_2 = \rho \quad \wedge \quad v_1 \cdot v_2 = v \quad \wedge \quad g \overset{\pi}{\underset{\rho_1}{\mapsto}} v_1 * g \overset{\pi}{\underset{\rho_2}{\mapsto}} v_2}$$

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \;\triangleq\; \exists (v_i)\; \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \wedge \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Two tree shares:

- π: permission (what can we do...)
- ρ: location (...to which part)

Composed value:

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

$$\frac{\rho_1 \oplus \rho_2 = \rho \qquad v_1 \cdot v_2 = v}{g \overset{\pi}{\underset{\rho_1}{\mapsto}} v_1 * g \overset{\pi}{\underset{\rho_2}{\mapsto}} v_2 \vdash g \overset{\pi}{\underset{\rho}{\mapsto}} v}$$

$$\frac{}{g \overset{\pi}{\underset{\rho}{\mapsto}} v \vdash \exists \rho_1, \rho_2, \quad \rho_1 \oplus \rho_2 = \rho \quad \wedge \quad g \overset{\pi}{\underset{\rho_1}{\mapsto}} v * g \overset{\pi}{\underset{\rho_2}{\mapsto}} 1}$$

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \quad \triangleq \quad \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \wedge \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Two tree shares:

- π: permission (what can we do...)
- ρ: location (...to which part)

Composed value:

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

$$\frac{\rho_1 \oplus \rho_2 = \rho \qquad v_1 \cdot v_2 = v}{g \overset{\pi}{\underset{\rho_1}{\mapsto}} v_1 * g \overset{\pi}{\underset{\rho_2}{\mapsto}} v_2 \vdash g \overset{\pi}{\underset{\rho}{\mapsto}} v}$$

$$\overline{g \overset{\pi}{\underset{\rho}{\mapsto}} v = \exists \rho_1, \rho_2, v_1, v_2, \quad \rho_1 \oplus \rho_2 = \rho \quad \wedge \quad v_1 \cdot v_2 = v \quad \wedge \quad g \overset{\pi}{\underset{\rho_1}{\mapsto}} v_1 * g \overset{\pi}{\underset{\rho_2}{\mapsto}} v_2}$$

Representation of ghost state

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v \quad \triangleq \quad \exists (v_i) \prod_{i \in \mathbb{N}_\rho} g_i \overset{\pi}{\mapsto} v_i \land \sum_{i \in \mathbb{N}_\rho} v_i = v$$

Two tree shares:

- π: permission (what can we do...)
- ρ: location (...to which part)

Composed value:

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

$$g \overset{\pi}{\underset{\rho}{\mapsto}} v = \exists \rho_1, \rho_2, v_1, v_2, \quad \rho_1 \oplus \rho_2 = \rho \quad \land \quad v_1 \cdot v_2 = v \quad \land \quad g \overset{\pi}{\underset{\rho_1}{\mapsto}} v_1 * g \overset{\pi}{\underset{\rho_2}{\mapsto}} v_2$$

Threads sharing memory

$$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$$

Threads sharing memory

$$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$$
$$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$$

Threads sharing memory

$$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$$
$$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \underset{\bullet}{\overset{\bullet}{\mapsto}} 0\}$$

```
V(s);
```

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$
$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \underset{\bullet}{\overset{\bullet}{\mapsto}} 0\}$

`V(s);`

$R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \underset{\bullet}{\overset{\circ}{\mapsto}} v$

Threads sharing memory

$\{s \stackrel{\bullet}{\boxdot\!\!\mapsto} R * x \stackrel{\bullet}{\mapsto} 0\}$
$\{s \stackrel{\bullet}{\boxdot\!\!\mapsto} R * x \stackrel{\bullet}{\mapsto} 0 * g \underset{\bullet}{\stackrel{\bullet}{\mapsto}} 0\}$

`V(s);`

$$R \triangleq \exists v\ x \stackrel{\bullet}{\mapsto} v * g \underset{\bullet}{\stackrel{◑}{\mapsto}} v$$

$\{s \stackrel{\bullet}{\boxdot\!\!\mapsto} R * g \underset{\bullet}{\stackrel{◐}{\mapsto}} 0\}$

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * x \overset{\bullet}{\mapsto} 0\}$
$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$

```
V(s);
```

$$R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$$

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$

`V(s);`

$R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

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

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$

```
V(s);
```

$R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

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

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 1 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 1\}$

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \underset{\bullet}{\mapsto} 0\}$

`V(s);` $\qquad R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

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

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 1 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 1\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$

`V(s);` $\qquad\qquad R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

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

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 1 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 1\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

```
P(s);
```

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * x \overset{\bullet}{\mapsto} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$

`V(s);` $\qquad R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

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

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 1 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 1\}$

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

`P(s);`

$\{s \overset{\bullet}{\boxdot\!\!\mapsto} R * x \overset{\bullet}{\mapsto} 2 * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \underset{\bullet}{\mapsto} 0\}$

`V(s);`

$$R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \underset{\bullet}{\overset{\RIGHTcircle}{\mapsto}} v$$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \underset{\bullet}{\overset{\LEFTcircle}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \underset{\LEFTcircle}{\overset{\LEFTcircle}{\mapsto}} 0 * g \underset{\RIGHTcircle}{\overset{\LEFTcircle}{\mapsto}} 0\}$

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

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \underset{\LEFTcircle}{\overset{\LEFTcircle}{\mapsto}} 1 * g \underset{\RIGHTcircle}{\overset{\LEFTcircle}{\mapsto}} 1\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \underset{\bullet}{\overset{\LEFTcircle}{\mapsto}} 2\}$

`P(s);`

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2 * g \underset{\bullet}{\overset{\LEFTcircle}{\mapsto}} 2\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2\}$

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$

`V(s);` $\qquad R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

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

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 1 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 1\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

```
P(s);
```

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2 * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2\}$

```
assert(x == 2);
```

Threads sharing memory

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} 0\}$
$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$

`V(s);` $\qquad R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

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

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 1 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 1\}$

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

`P(s);`

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} 2 * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} 2\}$

`assert(x == 2);`

$\{s \boxdot\!\!\overset{\bullet}{\mapsto} R * x \overset{\bullet}{\mapsto} 2\}$

Threads sharing memory

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 0 * g \overset{\bullet}{\underset{\bullet}{\mapsto}} 0\}$

```
V(s);
```

$R \triangleq \exists v\ x \overset{\bullet}{\mapsto} v * g \overset{\RIGHTcircle}{\underset{\bullet}{\mapsto}} v$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 0\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 0 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 0\}$

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

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\LEFTcircle}{\mapsto}} 1 * g \overset{\LEFTcircle}{\underset{\RIGHTcircle}{\mapsto}} 1\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

```
P(s);
```

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2 * g \overset{\LEFTcircle}{\underset{\bullet}{\mapsto}} 2\}$

$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2\}$

```
assert(x == 2);
```

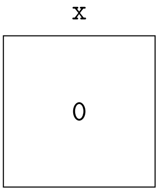
$\{s \overset{\bullet}{\boxdot\!\!\rightarrow} R * x \overset{\bullet}{\mapsto} 2\}$

safe!

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

Thread 1 | Shared resource | Thread 2

```
x = 0;
```

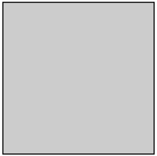


… we catch back on Nanevski's subjective views

Thread 1 | Shared resource | Thread 2

```
x = 0;
V(s);
```

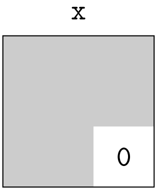
x



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

Thread 1 | Shared resource | Thread 2

```
x = 0;
V(s);
```




```
P(s);
```

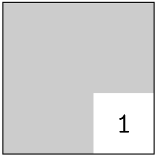
… we catch back on Nanevski's subjective views

Thread 1

```
x = 0;
V(s);
```

Shared resource

x



Thread 2

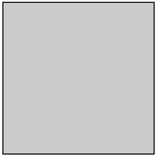
```
P(s);
x++;
```

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

Thread 1	Shared resource	Thread 2

```
x = 0;
V(s);
```

x



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

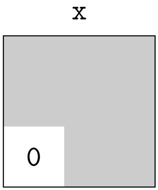
... we catch back on Nanevski's subjective views

Thread 1

```
x = 0;
V(s);

P(s);
```

Shared resource



Thread 2

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

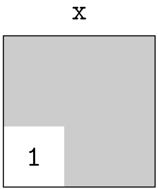
… we catch back on Nanevski's subjective views

Thread 1

```
x = 0;
V(s);

P(s);
x++;
```

Shared resource



Thread 2

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

… we catch back on Nanevski's subjective views

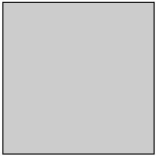
Thread 1

```
x = 0;
V(s);

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

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);

P(s);
```

Shared resource

x

0
$\oplus$
1 1

Thread 2

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

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

Thread 1	Shared resource	Thread 2
`x = 0;` `V(s);`		
	x $\boxed{2}$	

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

`P(s);`		

… we catch back on Nanevski's subjective views

Thread 1	Shared resource	Thread 2
`x = 0;` `V(s);`		
	x $\boxed{2}$	

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

`P(s);` `assert(x == 2);`		

… we catch back on Nanevski's subjective views

Thread 1 | Shared resource | Thread 2

Thread 1:

```
x = 0;
V(s);

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

P(s);
assert(x == 2);
```

Shared resource:

x

$\boxed{2}$

Thread 2:

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

Safe!

Summary

```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

```
#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

```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;
}
```

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

Summary

```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;
}
```

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

Summary

```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;
}
```

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)

Summary

```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;
}
```

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)

Summary

```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;
}
```

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

```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;
}
```

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?

Summary

```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;
}
```

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

```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;
}
```

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

```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;
}
```

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)
   $\simeq$ our ghost state

Program patterns

We can do:

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

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.

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!

$$g \xmapsto[\rho]{\pi} v$$

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