The Concurrent CMinor project

Logic for low-level concurrent imperative languages

Francesco Zappa Nardelli[1]

1. INRIA Rocquencourt

But this talk is about what Andrew Appel, Sandrine Blazy, and Aquinas Hobor taught me.

Joint research effort

Andrew Appel
Princeton U.

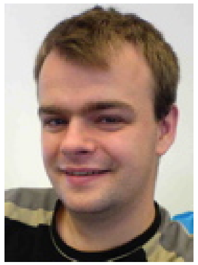
Sandrine Blazy
ENSIIE

Aquinas Hobor
Princeton U.

Adriana Compagnoni
Stevens Tech.

me
INRIA Rocq

...and some people that give us good advices:

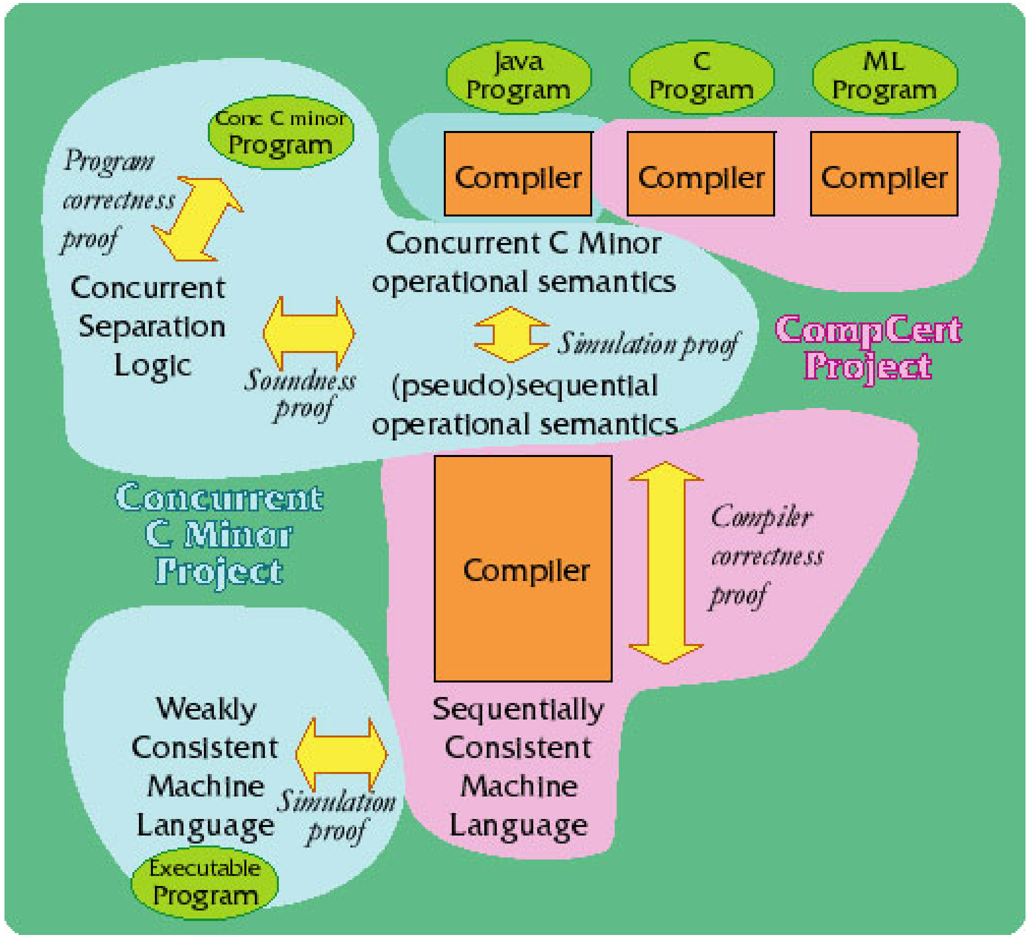


Matthew Parkinson
Cambridge U.

Peter O'Hearn
London U.

Xavier Leroy
INRIA Rocq

The big picture



Aim: connect machine-verified source programs in sequential and concurrent programming languages to machine-verified optimizing compilers.

CMinor for the C programmer

CMinor is a low-level imperative language. It looks like C with some restriction:

- no operator overloading nor implicit conversions;
- size of load and stores explicit;
- no general `goto` (but multi-exit loops);
- functions comprise a type signature and a declaration of how many bytes of stack-allocated space they need;
- variables do not reside in memory and their address cannot be taken - however, the CMinor producer can explicitly stack-allocate some data.

However CMinor is not a toy language

- Reading and writing from/to memory are expressions.
- Realistic memory model that is byte- and word- addressable.
- Pointer arithmetic within any malloc'ed block is defined, but undefined between different blocks.
  - pointer values comprise abstract block number and `int` offset;
  - expressions can evaluate to Vundef without getting stuck.
- non-trivial control-flow and function semantics.

Separation logic in one slide

Hoare logic relates the logical descriptions (assertions) of the states of a machine before and after the execution of a command:

$$\{P\}\ c\ \{Q\}\ .$$

Separation logic extends Hoare logic, by allowing assertions to describe both the stack and the heap of the machine.

Key observation: separate program texts which work on separate sections of the store can be reasoned about independently.

$$\begin{aligned}(e_1 \mapsto e_2)(s,h) &= \text{dom } h = \{\text{eval } e_1\ s\} \wedge h(\text{eval } e_1\ s) = \text{eval } e_2\ s \\ (P * Q)(s,h) &= \exists h1, h2.\ h = h1 \oplus h2 \wedge P(s,h1) \wedge Q(s,h2)\end{aligned}$$

Embedding separation logic into Coq

Two alternatives when doing machine-checked proofs of imperative programs:

- implement Hoare logic directly in a logical framework such as Twelf or Isabelle;
- define the operators of Hoare logic inside an higher-order logic such as Coq or Isabelle/HOL.

Advocate for the two-level approach: most of the reasoning is not about memory cells but about the abstract objects that the data structure represent. Lemmas about those objects are most conveniently proved in a general purpose higher-order logic.

But we need lemmas and tactics to move between the two levels.

Hoare's pun

Hoare's notation confuses program expressions with logical formulas:

$$\{a = b \cdot 2 + 1\}\ b \leftarrow b \cdot 2\ \{a = b + 1\}\ .$$

Are assertions boolean expressions from the programming language?

$$\{P\}\ c\ \{Q\} \equiv \forall s.\ \text{evalb}\ P\ s \Rightarrow \exists s'.\text{exec}\ c\ s' \wedge \text{evalb}\ Q\ s'$$

Clever! But we need a non-trivial assertion language (eg. quantifications)! So, consider assertions as predicates on states:

$$\{P\}\ c\ \{Q\} \equiv \forall s.\ P\ s \Rightarrow \exists s'.\text{exec}\ c\ s' \wedge Q\ s'$$

Hoare's pun (ctd.)

With assertions-as-predicates we must write

$$\{\lambda s. sa = 2(sb) + 1\}\ b \leftarrow b \cdot 2\ \{\lambda s. sa = sb + 1\}$$

or

$$\{\text{evalb}\ (a = b \cdot 2 + 1)\}\ b \leftarrow b \cdot 2\ \{\text{evalb}\ (a = b + 1)\}\ .$$

However, we have the expressive power to write quantified formulas:

$$\{\lambda s. \exists y. y = 2(sa) \wedge sb = y + 1\}\ b \leftarrow b \cdot 2\ \{\lambda s. \exists z. \text{evalb}\ (b = a + 1)\ s\}$$

Clumsy? No, in a tactical theorem prover

A typical situation, requires proving $\forall s. P s \Rightarrow Q s$, where P is the conjunction of several P_i. With lemmas and tactics we can break up the goal into subgoals such as:

$$\frac{\begin{array}{l} s : State \\ H_1 : P_1 \\ \vdots \\ H_n : P_n \end{array}}{Q_1}$$

and these can be proved using the normal lemmas and tactics in the prover's library.

Separation logic is different

Consider a goal such as

$$\forall sh. As \land (P * Q * R)sh \Rightarrow (Bs \land Ush) * Vsh$$

With routine tactics we obtain

$$\frac{\begin{array}{l} s : \text{Store} \quad h : \text{Heap} \\ H_1 : As \\ h_p : \text{Heap}\ h_q : \text{Heap}\ h_r : \text{Heap}\ h_{pq} : \text{Heap} \\ H_{pq} : h_{pq} = h_p \cup h_q \qquad H_h : h = h_{pq} \cup h_r \\ H_{pq'} : h_p \cap h_q = \{\} \qquad H_{h'} : h_{pq} \cap h_r = \{\} \\ H_2 P s h_p \qquad H_3 : Qsh_q \qquad H_4 : Rsh_r \end{array}}{Bs \land \exists h_u h_v . H = h_u \cup h_v \land h_u \cap h_v = \{\} \land Ush_u \land Vsh_v}$$

Tactics for separation logics

Disgusting! We want to hide the explicit manipulation of the heap: after all this is what separation logic is about. And...

- purely mathematical reasoning should proceed naturally;
- Hoare-triple reasoning should proceed naturally;
- there should be natural transitions between the two levels.

Is this possible? Yes! Demo...

Soundness of our (sequential) separation logic

Leroy compiler is proved correct with respect to a big-step semantics. This is not suitable for our purposes, as we want to reason about concurrent programs. We also want to support non-trivial control flow (avoid trace semantics), and avoid all search rules...

- define a continuation based semantics;
- prove that it is equivalent to Leroy big-step semantics;
- prove the soundness of the logic with respect to it.

The sequential machine

$$\Psi \vdash ((\sigma, \kappa), m) \mapsto ((\sigma', \kappa'), m')$$

- a global state:
  - a memory m;
  - a program (mapping function names to function bodies) Ψ;
- a closure:
  - a local state σ (definining the stack pointer sp, the environment ρ, and *the view of the world* w);
  - the current continuation κ.
- the continuations: Kstop | $s \cdot \kappa$ | Kblock κ | Kcall (...) κ

Worlds

A world gives permission to take actions on memory locations. For instance, in a world w such that:

$$w \vdash l_1 \wedge w \vdash l_2$$

the machine can read and modify the cells l_1 and l_2 while

the machine is stuck if it accesses another cell l_3.

For now, a world looks like a *footprint* and keeps track of the memory allocated to the current computation.

Interpretation of separation logic

- A continuation $k = (\sigma, \kappa)$ is stuck if $\kappa \neq$ Kstop and there does not exist k' such that $k \mapsto^* k'$.

- *A sequential state is safe if it cannot reach a stuck state.*

- an assertion P guards a control κ ($P \Box \kappa$) if whenever P holds κ is safe:

$$P \Box \kappa \quad \equiv \quad \forall \sigma, m.\ \sigma, m \vdash P \Rightarrow \mathsf{safe}((\sigma, \kappa), m)\ .$$

- Hoare triple:

$$\{P\}\ s\ \{Q\} \quad \equiv \quad \forall \kappa.\ Q \Box \kappa \Rightarrow P \Box s \cdot \kappa$$

Soundness and program safety

- Prove many lemmas that show when it is safe to conclude $\{P\}\ s\ \{Q\}$:

$$\forall P.\ \{P\}\ \textsf{skip}\ \{P\} \qquad \frac{\{P\}\ s\ \{Q\} \wedge (\forall \sigma, m.\ R \sigma m \Rightarrow P \sigma m)}{\{R\}\ s\ \{Q\}}$$

- Using the lemmas, showing that

$$\{\textsf{True}\}\ \textsf{call}\ \mathrm{main}\ \{\textsf{True}\}$$

for a program Ψ is enough to ensure that Ψ is safe.

Concurrent CMinor

Concurrent CMinor extends CMinor with thread creation (`spawn`), shared memory and semaphores.

Semaphores are allocated on the heap, and can be dinamically created.

Concurrent Separation Logic

O'Hearn ideas:

- if the precondition of a statement says $l \mapsto e$, then that statement has exclusive access to location l;
- each lock protects some memory locations: $\{P\}$ lock l $\{P * R_l\}$ where R_l is the lock invariant (an assertion that describes the locations protected by the lock).
- we can prove that well-synchronised programs are safe!

A realistic Concurrent Separation Logic

O'Hearn concurrent separation logic has several drawbacks (locks cannot be dynamically created, etc), but most importantly...

...it supposes that locks are shared variables allocated on the stack!

We designed a concurrent separation logic that:

- support a realistic memory model;
- allows creation and disposal of resources;
- allows concurrent read accesses using partial ownership:
  - writing a cell requires 100% of ownership, but reading only needs a fraction;
  - it holds that $l \mapsto_{100\%} v = l \mapsto_{50\%} v * l \mapsto_{50\%} v$.

The concurrent machine

$$\Psi \vdash \Omega, (\sigma_1, \kappa_1) \cdots (\sigma_n, \kappa_n), m, w_g \;\Rightarrow\; \Omega', (\sigma'_1, \kappa'_1) \cdots (\sigma'_n, \kappa'_n), m', w'_g$$

- the scheduler Ω is a list of natural numbers;
- there is a list of threads (σ_i, κ_i)
  - new continuations: Klock $e \cdot \kappa$ | Kunlock $e \cdot \kappa$ | Kfork $e\ el \cdot \kappa$;
- m is the memory;
- the global world specifies the invariants of unheld resources (next slide).

Worlds, refined

In the concurrent machine a world w:

- grants partial or total acces to a resource:

$$w \vdash \text{read } l_1 \;\wedge\; w \vdash \text{write } l_2$$

- specifies lock invariants:

$$w \vdash \text{lock } l \;\; \text{with } R$$

- specifies the state of locks:

$$w \vdash \text{hold } l$$

(A model can be built using stratification as in Appel, Méllies, Richards, Vouillon).

The behaviour of the concurrent machine

$$\Psi \vdash \Omega, (\sigma_1, \kappa_1) \cdots (\sigma_n, \kappa_n), m, w_g \;\Rightarrow\; \Omega', (\sigma'_1, \kappa'_1) \cdots (\sigma'_n, \kappa'_n), m', w'_g$$

A few cases... For the thread indicated by the scheduler Ω:

- if κ is sequential, run the thread non-preemptively until either
  - if becomes Klock $e \cdot \kappa$ | Kunlock $e \cdot \kappa$ | Kfork $e\ el \cdot \kappa$;
  - it becomes Kstop.

- if $\kappa =$ Klock e and the lock is available:
  - lock e;
  - transfer the permissions described in the invariant to the thread;
  - return to the thread list with κ.

Safety of the concurrent machine

- A concurrent state is safe if for any scheduler, we cannot reach a stuck state.
- This implies both noninterference and that all lock invariants are obeyed.
- *But how can we prove that a machine is safe?*
  - We want to state and prove correctness properties for threads (and these should combine to provide safety for the whole machine);
  - Reasoning sequentially is hard enough: hide as much of the concurrent machine as possible.

An oracular machine

$$\Psi \vdash (\omega, (\sigma, \kappa), m) \leadsto (\omega', (\sigma', \kappa'), m')$$

Ideas:

- for sequential actions, the machine behaves as the sequential machine;
- for lock, unlock or fork, the machine consults the oracle ω to see the state of the machine when this thread is scheduled again.

The oracle ω simply simulates the concurrent machine until the thread is called again.

Oracular safety

- A continuation $k = (\sigma, \kappa)$ is stuck if $\kappa \neq$ Kstop and there does not exist k' such that $k \mapsto^* k'$, *or κ is concurrent and the oracle tells us that we cannot continue*.

- *A state is safe if* for all oracles *it cannot reach a stuck state.*

- an assertion P guards a control κ ($P \Box \kappa$) if whenever P holds κ is safe:

$$P \Box \kappa \quad \equiv \quad \forall \sigma, m.\ \sigma, m \vdash P \Rightarrow \mathsf{safe}((\sigma, \kappa), m)\ .$$

- Hoare triple:

$$\{P\}\ s\ \{Q\} \quad \equiv \quad \forall \kappa.\ Q \Box \kappa \Rightarrow P \Box s \cdot \kappa$$

Recycling sequential reasoning

- Prove many lemmas that show when it is safe to conclude $\{P\}\ s\ \{Q\}$:

$$\forall \Psi, R.\ \{\text{lock } l \text{ with } R\}\ \text{lock } l\ \{\text{lock } l \text{ with } R * \text{hold } l * R\}$$

- Since ω does not change for any sequential instruction, all of the old proofs about purely sequential rules are reusable.

- Using the lemmas, we can prove correctness and safety properties for our threads when executed on the oracular machine.

- *Conjecture*: if every thread running on a machine is oracular safe, then the whole machine is safe.

Directions of the Concurrent CMinor project

- Show that our model of concurrency is reasonable for a modern machine:
  - *preemption*
  - *weak memory model*
- tactics for concurrent separation logic:
  - *automation*
  - *concurrency*
- relationship with lock inference algorithms
- correct compiler from Featherweight Java to CMinor.