Capabilities, MMIO and Robust Safety

March 10, 2020

1 Memory Mapped I/O: Operational Semantics

The proposal is to simply represent read and writes to memory-mapped IO addresses as events in a trace. This says nothing a priori about devices that might be connected to these IO regions. In particular, without additional assumptions, reading a byte from a memory-mapped region just returns an arbitrary value.

If at some point we want to reason under the assumption that we are connected to a specific device that reacts in a specific way, we can express that as an extra Separation Logic assertion, that we assume as a pre-condition, and that restricts the set of different traces that we might observe.

$$\begin{array}{lcl}
\text{EventTy} & := & \text{IOWrite} \mid \text{IORead} \\
\text{Event} & := & \text{EventTy} \times \text{Addr} \times \mathbb{Z} \\
\text{Trace} & := & \text{list Event} \\
\text{State} & := & \underbrace{\text{Reg} \times \text{Mem}}_{\text{old state}} \times \text{Trace}
\end{array}$$

Values of type State represent the state of a configuration in the small-step operational semantics. In this setup, we assume the whole semantics to be parameterized by the range of memory-mapped addresses: MMIO.

$$\mathsf{MMIO} \quad := \quad [\mathsf{MMIO}_b, \mathsf{MMIO}_e)$$

NB: An alternative presentation would be to make this range an immutable part of State. It probably does not really matter in the end, we should do whatever makes our life easier for instantiating Iris with the semantics.

In the (current) operational semantics without MMIO, the operational semantics of the `Load` instruction is:

$$\textsc{Load}\quad \frac{r[\text{src}] = (p, g, b, e, a) \qquad \text{readAllowed}\ p \qquad a \in [b, e)}{(r, m) \xrightarrow{\texttt{Load dst src}} (r[\text{dst} := m[a]], m)}$$

With MMIO, we obtain two rules:

$$\frac{r[\text{src}] = (p, g, b, e, a) \qquad \text{readAllowed}\, p \qquad a \in [b, e) \qquad a \notin \mathsf{MMIO}}{(r, m, t) \xrightarrow{\text{Load dst src}} (r[\text{dst} := m[a]], m, t)}\ \textsc{MemLoad}$$

$$\frac{r[\text{src}] = (p, g, b, e, a) \qquad \text{readAllowed}\, p \qquad a \in [b, e) \qquad a \in \mathsf{MMIO}}{(r, m, t) \xrightarrow{\text{Load dst src}} (r[\text{dst} := x], m, (\text{IORead}, a, x) :: t)}\ \textsc{IOLoad}$$

Notice how in the second rule, we read an arbitrary integer x, and record it in the trace.

Remark: With this presentation, the specification of the "machine" is very much decoupled from the model of the devices it might be communicating with. This is a good thing, I think.

Nevertheless, one could consider an alternative presentation where the model of the devices is more tightly integrated with the semantics of the machine. For instance, one could make the operational semantics parameterized with the devices' model, where each device is modeled as having some internal state, the ability to react on reads or writes, or to perform an internal step. Then, the operational semantics would either step the usual way, or whenever a device steps.

I believe this would be somewhat similar to the semantics of I/O system calls through a foreign function interfaces as formalized in CakeML [FPK$^+$18].

Remark: Alix says that the proposed style of operational semantics (using a trace) is close to the semantics of `volatile` as in CompCert—which is also how memory mapped addresses seem to be exposed to C compilers in practice. So this is probably a good sign.

2 Separation Logic Resources

We added a trace as part of the state, so we wish to also expose it as a Separation Logic assertion.

The simplest way to do that is to directly expose the complete trace. In that case, the state interpretation additionally holds a resource for the trace:

$$\text{state\_interp}\ (r, m, t) := \ldots * \boxed{\bullet\, t}^{\gamma_T}$$

The user then works with assertions of the form $\boxed{\circ\, t'}^{\gamma_T}$. Such an assertion is not duplicable, and grants full ownership over the trace. In particular, it allows one to *update* the trace by emitting events, i.e. by performing I/O operations.

The wp-rules for `Load` and `Store` need to be updated consequently. For instance, the rule for a `Load` reading the integer x on a MMIO address a now requires $\boxed{\circ\, t}^{\gamma_T}$ in the pre-condition (for some t), and provides $\boxed{\circ\, (\mathrm{IORead}, a, x) :: t}^{\gamma_T}$ in the post-condition.

The corresponding resource algebra is $\textsc{Auth}(\textsc{Ex}(\mathrm{Trace}))$. A few relevant rules are:

$$\begin{array}{l}
\boxed{\bullet\, t}^{\gamma_T} * \boxed{\circ\, t'}^{\gamma_T} \mathrel{-\!\!*} \lceil t = t' \rceil \\
\boxed{\circ\, t}^{\gamma_T} * \boxed{\circ\, t'}^{\gamma_T} \mathrel{-\!\!*} \mathsf{False} \\
\boxed{\bullet\, t}^{\gamma_T} * \boxed{\circ\, t}^{\gamma_T} \Rrightarrow\!\!\ast\ \boxed{\bullet\, t'}^{\gamma_T} * \boxed{\circ\, t'}^{\gamma_T}
\end{array}$$

Remark: This is very coarse-grained: either one has the full ownership for performing I/O and reasoning about it, or one cannot know anything about it.

A first extension could be to allow observing prefixes of the trace (since events can only be appended to the trace). The observation that the trace has a given prefix would be duplicable.

Another extension, that seems very useful for modular reasoning, would be to allow splitting the trace along separate range of addresses. Concretely, a trace containing events about the range of MMIO addresses $[a, c)$ could be split into two traces, granting ownership over events on addresses $[a, b)$ and $[b, c)$ respectively. Note that recombining these two traces would only yield some unspecified interleaving of the events from the two traces, and not necessarily yield the original trace.

One application of this second extension could be the verification of an example involving "multiplexed" I/O, where two separate parts of the code are granted separate ownership over separate MMIO addresses. These two separate pieces of code would be verified separately; then, in the end, one could prove that one gets *some* interleaving of all the events emitted by both components.

3 Toplevel Theorem (*à la* OCPL)

Let us start with a brief recap of the toplevel "Robust Safety" theorem for OCPL itself (and its key ingredients), then the extension (by Thomas) of OCPL to include a `print` capability, and finally move to the capability machine setting.

3.1 OCPL

The RobustSafety theorem of OCPL is as follows:

$$\frac{e \text{ closed} \qquad \{\mathsf{True}\}\, e\, \{x.\ \mathsf{lowval}\ x\} \qquad \begin{array}{c} C \in \mathit{AdvCtx} \\ (C[e]); (\emptyset, \mathsf{OK}) \longrightarrow^* T'; (h', g') \end{array}}{g' = \mathsf{OK}}$$

For any closed expression e, if e has been verified to only return low values, then running e wrapped in an adversarial context C from an initial state, then we can observe that every reachable state is good ($g' = \mathsf{OK}$ means that no assertion has failed in e).

$C \in \mathit{AdvCtx}$ means that C cannot contain assertions (otherwise one could trivially contradict the theorem by taking $C[\cdot] = \mathsf{assert\ false}$), and cannot contain references to raw memory locations (otherwise one could directly access e's private state, invalidating the local state encapsulation mechanisms).

$\mathsf{lowval}\ x$ intuitively means that x cannot be used to directly access private (or "high") locations. It is formally defined using a logical relation "$\mathsf{lift}\ \Psi\ v$", which more generally asserts that the value v only gives direct access to locations that satisfy the predicate Ψ.

$$\begin{array}{lcl}
\mathsf{lift}\ \Psi\ (\mathsf{rec}\ f\ x.\ e) & \triangleq & \triangleright \forall v.\ \{\mathsf{lift}\ \Psi\ v\}\, e[v/x, \mathsf{rec}\ f\ x.\ e/f]\, \{y.\ \mathsf{lift}\ \Psi\ y\} \\
\mathsf{lift}\ \Psi\ (v_1, v_2) & \triangleq & \triangleright(\mathsf{lift}\ \Psi\ v_1, \mathsf{lift}\ \Psi\ v_2) \\
\mathsf{lift}\ \Psi\ \ell & \triangleq & \Psi\ \ell \\
\dots & &
\end{array}$$

Then, $\mathsf{lowval}\ x$ is defined as $\mathsf{lift\ lowloc}\ x$, where lowloc is a predicate characterizing the region of memory containing "low locations" (distinct from the other region containing "high locations").

3.2 OCPL with `print`

In Thomas' extension of OCPL, a new value Out is added, denoting an "output object capability", as well as a print primitive, where $\mathsf{print\ Out}\ v$ effectively "prints" the value v, i.e. adds it to the trace of printed values.

One then wants to be able to encapsulate the use of Out, for instance by defining object capabilities that enforce some invariants on the values being printed. In that setting, Out is considered as a "high" value:

FIXME: Thomas, I realize that this is less general that what you have (I think?), since in your case Out *can* be low, if the trace predicate is equivalent to True. But I don't know how to state that in a setting where the predicate is not directly embedded in the definitions...

$$\mathsf{lift}\ \Psi\ \mathsf{Out} \quad \triangleq \quad \mathsf{False}$$

The theorem then becomes:

$$\frac{C \in \mathit{AdvCtx} \qquad e \text{ closed} \qquad \boxed{\exists t.\ \boxed{\circ\, t}^{\gamma_T} * P(t)}^{\iota} \vdash \{\mathsf{True}\}\, e\, \{x.\ \mathsf{lowval}\ x\} \\ (C[e]); (\emptyset, \mathsf{OK}); \emptyset \longrightarrow^{*} T'; (h', g'); t}{g' = \mathsf{OK} \wedge P(t)}$$

That is, if e has been verified under the assumption that the predicate P holds as a trace invariant, then executing e in an adversarial context yields a trace that does satisfy P.

$C \in \mathit{AdvCtx}$ also has to be extended to forbid C from containing Out.

3.3 Capability Machine with MMIO

Similarly to lift, we can generalize the existing logical relation to thread a constraint on directly accessible memory locations. One would parameterize the value, expression and register relations ($\mathcal{V}$, ER and RR) with a predicate Ψ on addresses.

Then, for any permission p that includes either the R or W bit, $\mathcal{V}$ is extended as follows:

$$\mathcal{V}^{\Psi}(p, g, b, e, a) \triangleq \underbrace{\ldots}_{\text{as before}} * \ \lceil \forall a' \in [b, e).\ \Psi(a') \rceil$$

Then, $\mathcal{E}^{\Psi}$ and $\mathcal{R}^{\Psi}$ are simplify defined by threading the extra Ψ parameter through the existing definition.

Finally, for our relation to characterize "low values" that do not give direct access to MMIO addresses, one would instantiate Ψ with a predicate $\overline{\mathsf{MMIO}}$ that excludes memory mapped addresses:

$$\overline{\mathsf{MMIO}}(a) \triangleq a \notin \mathsf{MMIO}$$

Then, $\mathcal{V}^{\overline{\mathsf{MMIO}}}$ is similar to the lowval predicate of OCPL. Intuitively, a value in the relation does not directly point to memory-mapped locations, and also does not leak memory-mapped location through registers when interacting with other code.

Remark: As stated above, the "generalized" value relation is in fact not very useful for instantiation of Ψ other than $\overline{\mathsf{MMIO}}$. Indeed, even if Ψ *does* allow referring to addresses in the MMIO region, the value relation does not grant any corresponding resources. The fix would be to use the generalized trace resources mentioned previously, and grant ownership for the part of the trace corresponding to addresses in $[b, e) \cap \mathsf{MMIO}$.

Then, the toplevel theorem would be:

$$\frac{\begin{array}{c}\boxed{\exists t.\ \boxed{\circ\, t}^{\gamma_T} * P(t)}^{\iota} \vdash \mathcal{V}^{\overline{\mathsf{MMIO}}}(E, g, b, e, a) \\ (r[\mathrm{PC} := (E, g, b, e, a)], m, \emptyset) \longrightarrow^{*} (r', m', t)\end{array}}{P(t)}$$

FIXME: We need to add refine the assumptions of the theorem so that it has a chance of being true. One needs to add either a “syntactic condition” over thee state of the initial memory, or link with an initial boot-code and assume require the boot-code has a specification that entails the syntactic condition.

References

[FPK$^+$18] Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. Program verification in the presence of I/O: Semantics, verified library routines, and verified applications. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments (VSTTE)*, volume 11294 of *Lecture Notes in Computer Science*. Springer, 2018.