A deeper understanding of the deep frame axiom

(frame rules for higher order store)

Billiejoe (Nathaniel) Charlton and Bernhard Reus
University of Sussex

Setting the scene

- Higher order store means that mutable state, such as the heap, can be used to store code/commands/procedures

Setting the scene

- Higher order store means that mutable state, such as the heap, can be used to store code/commands/procedures

- Languages with higher order store can serve as a foundation to model e.g.

  - dynamic loading, runtime code generation, “hot update”, self-modifying code

Setting the scene

- Higher order store means that mutable state, such as the heap, can be used to store code/commands/procedures
- Languages with higher order store can serve as a foundation to model e.g.
  - dynamic loading, runtime code generation, "hot update", self-modifying code
- We are interested in logical reasoning for such languages
  - we use separation logic (a variant of Hoare logic)

Setting the scene

- Higher order store means that mutable state, such as the heap, can be used to store code/commands/procedures

- Languages with higher order store can serve as a foundation to model e.g.
  - dynamic loading, runtime code generation, "hot update", self-modifying code

- We are interested in logical reasoning for such languages
  - we use separation logic (a variant of Hoare logic)

- Particularly interested in hidden state

Hidden state

- How can we reason about hidden state effectively and soundly?

  How to reason about invocation of a higher order procedure
  when one of the arguments is a procedure with its own state

- Hidden state is really great: it leads to modular programs and modular proofs
  - But it's also tricky to reason about correctly

A program featuring hidden state

written in a minimal language with higher order store

$$
\begin{aligned}
&\mathsf{let}\ runIt = \mathsf{new}\ `\lambda f.\ \mathsf{eval}[f]()\text{'}\ \mathsf{in}\\
&\mathsf{let}\ f_1 = \mathsf{new}\ `\mathsf{skip}\text{'}\ \mathsf{in}\\
&\mathsf{let}\ ctr = \mathsf{new}\ 0\ \mathsf{in}\\
&\mathsf{let}\ f_2 = \mathsf{new}\ `[ctr] := [ctr] + 1\text{'}\ \mathsf{in}\\
&\quad \mathsf{eval}\ [runIt](f_2)\ ;\\
&\quad \mathsf{free}\ ctr\ ;\\
&\quad \mathsf{eval}\ [runIt](f_1)
\end{aligned}
$$

A program featuring hidden state

written in a minimal language with higher order store

$$
\begin{array}{l}
\mathsf{let}\ \ runIt = \mathsf{new}\ `\lambda f.\ \mathsf{eval}[f]()\text{'}\ \mathsf{in} \\
\mathsf{let}\ \ f_1 = \mathsf{new}\ `\mathsf{skip}\text{'}\ \mathsf{in} \\
\mathsf{let}\ \ ctr = \mathsf{new}\ 0\ \mathsf{in} \\
\mathsf{let}\ \ f_2 = \mathsf{new}\ `[ctr] := [ctr] + 1\text{'}\ \mathsf{in} \\
\quad \mathsf{eval}\ [runIt](f_2)\ ; \\
\quad \mathsf{free}\ \ ctr\ ; \\
\quad \mathsf{eval}\ [runIt](f_1)
\end{array}
$$

The ctr cell is “hidden state” in this call

This program is completely safe – it cannot crash.

Can we prove this?

Problem considered in this talk

PROBLEM:

- How can we reason about hidden state effectively and soundly?
- A logical axiom, called the Deep Frame Axiom, has been previously proposed for reasoning about hidden state (*Schwinghammer et al (CSL, 2008)*)

  - at first glance, appears to be natural and exactly what we need

Problem considered in this talk

PROBLEM:

- How can we reason about hidden state effectively and soundly?
- A logical axiom, called the Deep Frame Axiom, has been previously proposed for reasoning about hidden state (*Schwinghammer et al (CSL, 2008)*)

  - at first glance, appears to be natural and exactly what we need
- **Unfortunately it isn't sound!**

  - we can use it to prove correctness of a crashing program

Problem considered in this talk

PROBLEM:

- How can we reason about hidden state effectively and soundly?
- A logical axiom, called the Deep Frame Axiom, has been previously proposed for reasoning about hidden state (*Schwinghammer et al (CSL, 2008)*)

  - at first glance, appears to be natural and exactly what we need
- **Unfortunately it isn't sound!**

  - we can use it to prove correctness of a crashing program

SOLUTION:

- Propose a sound specification idiom which we can use instead

  - using second order logic (quantification over assertions)

Nested Hoare triples

We can reason about higher order store using a logic with nested triples, based on *Schwinghammer et al, CSL, 2008*. For example, consider our code for *runIt*:

$$\lambda f.\ \mathsf{eval}[f]()$$

This code can be specified by a Hoare triple:

$$\forall f.\ \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \ \{\mathsf{emp}\}\ \} \end{array}$$

The code is higher order so pre- and post-conditions contain Hoare triples.

$\{ \ \textsf{emp} \ \}$

$\textsf{let } runIt = \textsf{new } `\lambda f.\ \textsf{eval}[f]()\text{'} \textsf{ in}$
$\textsf{let } f_1 = \textsf{new } `\textsf{skip}\text{'} \textsf{ in}$
$\textsf{let } ctr = \textsf{new } 0 \textsf{ in}$
$\textsf{let } f_2 = \textsf{new } `[ctr] := [ctr] + 1\text{'} \textsf{ in}$
$\quad \textsf{eval } [runIt](f_2) \ ;$
$\quad \textsf{free } ctr \ ;$
$\quad \textsf{eval } [runIt](f_1)$

$\{ \ \mathrm{True} \ \}$

$$\forall f. \quad \{\ f \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\}\ \}\ \ \lambda f.\ \mathsf{eval}[f]()\ \ \{\ f \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\}\ \}$$

$$\{\ \mathsf{emp}\ \}$$

$$\begin{array}{l}
\mathsf{let}\ \mathit{runIt} = \mathsf{new}\ `\lambda f.\ \mathsf{eval}[f]()\text{'}\ \mathsf{in} \\
\mathsf{let}\ f_1 = \mathsf{new}\ `\mathsf{skip}\text{'}\ \mathsf{in} \\
\mathsf{let}\ \mathit{ctr} = \mathsf{new}\ 0\ \mathsf{in} \\
\mathsf{let}\ f_2 = \mathsf{new}\ `[\mathit{ctr}] := [\mathit{ctr}] + 1\text{'}\ \mathsf{in} \\
\quad \mathsf{eval}\ [\mathit{runIt}](f_2)\ ; \\
\quad \mathsf{free}\ \mathit{ctr}\ ; \\
\quad \mathsf{eval}\ [\mathit{runIt}](f_1)
\end{array}$$

$$\{\ \mathrm{True}\ \}$$

$$\left\{ \mathit{runIt} \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \right\}$$

let f_1 = new ‘skip’ in
let ctr = new 0 in
let f_2 = new ‘$[ctr] := [ctr] + 1$’ in
  eval $[runIt](f_2)$;
  free ctr ;
  eval $[runIt](f_1)$

$$\{\ \mathrm{True}\ \}$$

$$\left\{ runIt \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\} \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\} \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \right\}$$

let f_1 = new ‘skip’ in ← {emp} skip {emp}
let ctr = new 0 in
let f_2 = new ‘$[ctr] := [ctr] + 1$’ in
  eval $[runIt](f_2)$;
  free ctr ;
  eval $[runIt](f_1)$

$$\{\ \mathrm{True}\ \}$$

$$
\left\{
\begin{array}{l}
runIt \mapsto \forall f. \begin{array}{c} \{\; f \mapsto \{\mathsf{emp}\} \cdot () \; \{\mathsf{emp}\} \;\} \\ \cdot(f) \\ \{\; f \mapsto \{\mathsf{emp}\} \cdot () \; \{\mathsf{emp}\} \;\} \end{array} \\
\star\ f_1 \mapsto \{\mathsf{emp}\} \cdot () \; \{\mathsf{emp}\}
\end{array}
\right\}
$$

$\mathsf{let}\ ctr = \mathsf{new}\ 0\ \mathsf{in}$

$\mathsf{let}\ f_2 = \mathsf{new}\ `[ctr] := [ctr] + 1\text{'}\ \mathsf{in}$

$\quad \mathsf{eval}\ [runIt](f_2)\ ;$

$\quad \mathsf{free}\ ctr\ ;$

$\quad \mathsf{eval}\ [runIt](f_1)$

$$\{\; \mathrm{True} \;\}$$

$$\left\{ \begin{array}{l} runIt \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot (f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \\ \star\ f_1 \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\} \\ \star\ ctr \mapsto 0 \end{array} \right\}$$

$$\begin{array}{l} \mathsf{let}\ f_2 = \mathsf{new}\ \text{‘}[ctr] := [ctr] + 1\text{’}\ \mathsf{in} \\ \quad \mathsf{eval}\ [runIt](f_2)\ ; \\ \quad \mathsf{free}\ ctr\ ; \\ \quad \mathsf{eval}\ [runIt](f_1) \end{array}$$

$$\{\ \mathrm{True}\ \}$$

$$\left\{ \begin{array}{l} \mathit{runIt} \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \\ \star\ f_1 \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\} \\ \star\ \mathit{ctr} \mapsto 0 \end{array} \right\}$$

$\mathsf{let}\ f_2 = \mathsf{new}\ `[\mathit{ctr}] := [\mathit{ctr}] + 1\text{'}\ \mathsf{in}$

$\quad \mathsf{eval}\ [\mathit{runIt}](f_2)\ ;$

$\quad \mathsf{free}\ \mathit{ctr}\ ;$

$\quad \mathsf{eval}\ [\mathit{runIt}](f_1)$

$$\{\ \mathit{ctr} \mapsto \_\ \}$$
$$[\mathit{ctr}] := [\mathit{ctr}] + 1$$
$$\{\ \mathit{ctr} \mapsto \_\ \}$$

$$\{\ \mathrm{True}\ \}$$

$$\left\{ \begin{array}{l} \qquad\qquad\qquad \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ runIt \mapsto \forall f. \qquad\qquad\qquad \cdot (f) \\ \qquad\qquad\qquad \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \star\ f_1 \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\} \\ \star\ ctr \mapsto 0 \\ \star\ f_2 \mapsto \{ctr \mapsto \_\}\ \cdot ()\ \{ctr \mapsto \_\} \end{array} \right\}$$

$$\begin{array}{l} \mathsf{eval}\ [runIt](f_2)\ ; \\ \mathsf{free}\ ctr\ ; \\ \mathsf{eval}\ [runIt](f_1) \end{array}$$

$$\{\ \mathrm{True}\ \}$$

$$\left\{ \begin{array}{l} runIt \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\}\ \} \end{array} \\ \star\ f_1 \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\} \\ \star\ ctr \mapsto 0 \\ \star\ f_2 \mapsto \{ctr \mapsto \_\} \cdot () \ \{ctr \mapsto \_\} \end{array} \right\}$$

$$\begin{array}{l} \mathsf{eval}\ [runIt](f_2)\ ; \\ \mathsf{free}\ ctr\ ; \\ \mathsf{eval}\ [runIt](f_1) \end{array}$$

$$\{\ \mathrm{True}\ \}$$

Now we have a mismatch ☹ because the code in *runIt* doesn't know about the counter cell.

What can we do??

“Deep” framing

We introduce an operator $\otimes$ for adding invariants to specifications:

$P \otimes I$ means, informally, add I to every pre- and post-condition in *P*, at all nesting levels.

“Deep” framing

We introduce an operator $\otimes$ for adding invariants to specifications:

$P \otimes I$ means, informally, add I to every pre- and post-condition in *P*, at all nesting levels.

E.g.

$$\left(\forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\}\ \} \end{array} \right) \otimes ctr \mapsto \_$$

“Deep” framing

We introduce an operator $\otimes$ for adding invariants to specifications:

$P \otimes I$ means, informally, add I to every pre- and post-condition in *P*, at all nesting levels.

E.g.

$$\left(\forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \right) \otimes\ ctr \mapsto \_$$

means

$$\forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp} \star ctr \mapsto \_\}\ \cdot ()\ \{\mathsf{emp} \star ctr \mapsto \_\} \star ctr \mapsto \_\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp} \star ctr \mapsto \_\}\ \cdot ()\ \{\mathsf{emp} \star ctr \mapsto \_\} \star ctr \mapsto \_\ \} \end{array}$$

Deep Frame Axiom and Rule

Two mechanisms for adding invariants to specifications were considered by *Schwinghammer et al (CSL, 2008)*

DEEP FRAME RULE

$$\frac{P}{P \otimes I}$$

DEEP FRAME AXIOM

$$P \quad \Rightarrow \quad P \otimes I$$

Deep Frame Axiom and Rule

Two mechanisms for adding invariants to specifications were considered by *Schwinghammer et al (CSL, 2008)*

$$\textsc{Deep Frame Rule} \quad \frac{P}{P \otimes I}$$

$$\textsc{Deep Frame Axiom} \quad P \Rightarrow P \otimes I$$

The Deep Frame Rule can add invariants to a specification

- but only at the top level of the proof

Deep Frame Axiom and Rule

Two mechanisms for adding invariants to specifications were considered by *Schwinghammer et al (CSL, 2008)*

Deep Frame Rule

$$\frac{P}{P \otimes I}$$

Deep Frame Axiom

$$P \quad \Rightarrow \quad P \otimes I$$

The Deep Frame Rule can add invariants to a specification

- but only at the top level of the proof

The Deep Frame Axiom is stronger

- can also be used inside pre- and post-conditions of triples

$$\left\{ \begin{array}{l} runIt \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \\ \star\ f_1 \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\} \\ \star\ ctr \mapsto 0 \\ \star\ f_2 \mapsto \{ctr \mapsto \_\}\ \cdot ()\ \{ctr \mapsto \_\} \end{array} \right\}$$

$$\begin{array}{l} \mathsf{eval}\ [runIt](f_2)\ ; \\ \mathsf{free}\ ctr\ ; \\ \mathsf{eval}\ [runIt](f_1) \end{array}$$

$$\{\ \mathrm{True}\ \}$$

Here is our mismatch again.

Let's use the Deep Frame Axiom to add $ctr \mapsto \_$ as an invariant to the specification for *runIt*

$$\left\{\begin{array}{l} runIt \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{ctr \mapsto \_\}\ \cdot ()\ \{ctr \mapsto \_\} \star ctr \mapsto \_\ \} \\ \cdot(f) \\ \{\ f \mapsto \{ctr \mapsto \_\}\ \cdot ()\ \{ctr \mapsto \_\} \star ctr \mapsto \_\ \} \end{array} \\ \star\ f_1 \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\} \\ \star\ ctr \mapsto 0 \\ \star\ f_2 \mapsto \{ctr \mapsto \_\}\ \cdot ()\ \{ctr \mapsto \_\} \end{array}\right\}$$

$$\begin{array}{l} \mathsf{eval}\ [runIt](f_2)\ ; \\ \mathsf{free}\ ctr\ ; \\ \mathsf{eval}\ [runIt](f_1) \end{array}$$

$$\{\ \mathrm{True}\ \}$$

Now we can reason about the call and we are happy ☺

The Deep Frame Axiom is unsound

- We've just seen why we want the Deep Frame Axiom
  - unfortunately it is not sound; only the weaker rule version is sound

The Deep Frame Axiom is unsound

- We've just seen why we want the Deep Frame Axiom

  - unfortunately it is not sound; only the weaker rule version is sound

- Because we can hide state, when doing a proof we might already have hidden some state to get the precondition!

  - So in general the code in *runIt* may have access to heap cells we don't know about

The Deep Frame Axiom is unsound

- We've just seen why we want the Deep Frame Axiom

  - unfortunately it is not sound; only the weaker rule version is sound

- Because we can hide state, when doing a proof we might already have hidden some state to get the precondition!

  - So in general the code in *runIt* may have access to heap cells we don't know about

- If the *runIt* code copies "outside" code into the hidden cells, things can go wrong:

  - The program will crash

  - But we can still prove it correct using the Deep Frame Axiom

$$\text{let } hidden = \text{new `skip' in}$$
$$\text{let } runIt = \text{new `}\lambda f \text{ . eval}[hidden]() \text{ ; } [hidden] := [f]\text{' in}$$

$$\text{let } f_1 = \text{new `skip' in}$$
$$\text{let } ctr = \text{new } 0 \text{ in}$$
$$\text{let } f_2 = \text{new `}[ctr] := [ctr] + 1\text{' in}$$
$$\quad \text{eval } [runIt](f_2) \text{ ;}$$
$$\quad \text{free } ctr \text{ ;}$$
$$\quad \text{eval } [runIt](f_1)$$

CRASH!

$$\{ \text{ emp } \}$$

let $hidden$ = new 'skip' in
let $runIt$ = new 'λf . eval$[hidden]()$; $[hidden] := [f]$' in

let f_1 = new 'skip' in
let ctr = new 0 in
let f_2 = new '$[ctr] := [ctr] + 1$' in
  eval $[runIt](f_2)$;
  free ctr ;
  eval $[runIt](f_1)$

$$\{ \text{ True } \}$$

$\{\ \mathsf{emp}\ \}$

$\mathsf{let}\ hidden = \mathsf{new}\ \text{‘}\mathsf{skip}\text{’}\ \mathsf{in}$

$\mathsf{let}\ runIt = \mathsf{new}\ \text{‘}\lambda f\ .\ \mathsf{eval}[hidden]()\ ;\ [hidden] := [f]\text{’}\ \mathsf{in}$

$$\left\{ \begin{array}{l} runIt \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \otimes hidden \mapsto \cdots \\ \star\ hidden \mapsto \cdots \end{array} \right\}$$

$\mathsf{let}\ f_1 = \mathsf{new}\ \text{‘}\mathsf{skip}\text{’}\ \mathsf{in}$

$\mathsf{let}\ ctr = \mathsf{new}\ 0\ \mathsf{in}$

$\mathsf{let}\ f_2 = \mathsf{new}\ \text{‘}[ctr] := [ctr] + 1\text{’}\ \mathsf{in}$

$\mathsf{eval}\ [runIt](f_2)\ ;$

$\mathsf{free}\ ctr\ ;$

$\mathsf{eval}\ [runIt](f_1)$

$\{\ \mathrm{True}\ \}$

$$\{\ \mathsf{emp}\ \}$$

$\mathsf{let}\ hidden = \mathsf{new}\ \text{`}\mathsf{skip}\text{'}\ \mathsf{in}$

$\mathsf{let}\ runIt = \mathsf{new}\ \text{`}\lambda f\ .\ \mathsf{eval}[hidden]()\ ;\ [hidden] := [f]\text{'}\ \mathsf{in}$

$$\left\{ runIt \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \right\}$$

$\mathsf{let}\ f_1 = \mathsf{new}\ \text{`}\mathsf{skip}\text{'}\ \mathsf{in}$

$\mathsf{let}\ ctr = \mathsf{new}\ 0\ \mathsf{in}$

$\mathsf{let}\ f_2 = \mathsf{new}\ \text{`}[ctr] := [ctr] + 1\text{'}\ \mathsf{in}$

$\mathsf{eval}\ [runIt](f_2)\ ;$

$\mathsf{free}\ ctr\ ;$

$\mathsf{eval}\ [runIt](f_1)$

$$\{\ \mathrm{True}\ \}$$

$$\{\ \mathsf{emp}\ \}$$

$\mathsf{let}\ hidden = \mathsf{new}\ \text{`}\mathsf{skip}\text{'}\ \mathsf{in}$

$\mathsf{let}\ runIt = \mathsf{new}\ \text{`}\lambda f\ .\ \mathsf{eval}[hidden]()\ ;\ [hidden] := [f]\text{'}\ \mathsf{in}$

$$\left\{ runIt \mapsto \forall f. \begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \right\}$$

$\mathsf{let}\ f_1 = \mathsf{new}\ \text{`}\mathsf{skip}\text{'}\ \mathsf{in}$

$\mathsf{let}\ ctr = \mathsf{new}\ 0\ \mathsf{in}$

$\mathsf{let}\ f_2 = \mathsf{new}\ \text{`}[ctr] := [ctr] + 1\text{'}\ \mathsf{in}$

$\mathsf{eval}\ [runIt](f_2)\ ;$

$\mathsf{free}\ ctr\ ;$

$\mathsf{eval}\ [runIt](f_1)$

$$\{\ \mathrm{True}\ \}$$

Obviously this is bad – we seem to need the Deep Frame Axiom but it is unsound.

How to resolve the problem?

Proposed solution

- We’ve seen two implementations for *runIt*:

  - one where adding invariants in axiom style is safe, another where it is not

Proposed solution

- We've seen two implementations for *runIt*:

  - one where adding invariants in axiom style is safe, another where it is not

- Thus, whether or not it is safe to add invariants must become part of the specification agreed between the *runIt* code and its clients.

Proposed solution

- We've seen two implementations for *runIt*:

  - one where adding invariants in axiom style is safe, another where it is not

- Thus, whether or not it is safe to add invariants must become part of the specification agreed between the *runIt* code and its clients.

- This can be expressed easily using second order logic:

$$\forall X.\,\forall f.\left(\begin{array}{c}\{\ f \mapsto \{\mathsf{emp}\}\ \cdot()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot()\ \{\mathsf{emp}\}\ \}\end{array}\right) \otimes X$$

Proposed solution

- We've seen two implementations for *runIt*:

  - one where adding invariants in axiom style is safe, another where it is not

- Thus, whether or not it is safe to add invariants must become part of the specification agreed between the *runIt* code and its clients.

- This can be expressed easily using second order logic:

$$\forall X.\, \forall f. \left(\begin{array}{c} \{\ f \mapsto \{\mathsf{emp}\} \cdot ()\ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\} \cdot ()\ \{\mathsf{emp}\}\ \} \end{array} \right) \otimes X$$

- With this idiom we can prove the correct program, but not the faulty one ☺

$$\left\{ runIt \mapsto \forall X.\, \forall f. \begin{pmatrix} \{\ f \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\}\ \} \\ \cdot(f) \\ \{\ f \mapsto \{\mathsf{emp}\} \cdot () \ \{\mathsf{emp}\}\ \} \end{pmatrix} \otimes X \right\}$$

$$\begin{array}{l} \mathsf{let}\ f_1 = \mathsf{new}\ \text{‘}\mathsf{skip}\text{’}\ \mathsf{in} \\ \mathsf{let}\ ctr = \mathsf{new}\ 0\ \mathsf{in} \\ \mathsf{let}\ f_2 = \mathsf{new}\ \text{‘}[ctr] := [ctr] + 1\text{’}\ \mathsf{in} \\ \quad \mathsf{eval}\ [runIt](f_2)\ ; \\ \quad \mathsf{free}\ ctr\ ; \\ \quad \mathsf{eval}\ [runIt](f_1) \end{array}$$

$$\{\ \mathrm{True}\ \}$$

We can easily prove this; just instantiate X with

$$ctr \mapsto \_$$

when you need to ☺

Remarks

- Commands specified with $\forall X. \cdots \otimes X$ may still have hidden state

  - and they may still use that hidden state for storing code

Remarks

- Commands specified with $\forall X. \cdots \otimes X$ may still have hidden state
  - and they may still use that hidden state for storing code
- Copying outside code into hidden state seems to be what is ruled out
  - We would like to be able to be more precise about this

Remarks

- Commands specified with $\forall X. \cdots \otimes X$ may still have hidden state
  - and they may still use that hidden state for storing code
- Copying outside code into hidden state seems to be what is ruled out
  - We would like to be able to be more precise about this
- In proofs we have done so far, using the $\forall X. \cdots \otimes X$ specification didn't generate much extra work
  - We would like to be more precise about this too

The end

Remarks

E.g. for the implementation of *runIt* which doesn't use hidden state, the extra $\forall X. \cdots \otimes X$ comes for free via the DFR:

$$
\dfrac{\dfrac{\forall f.\ \begin{array}{c}\{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \}\\ \lambda f.\ \mathsf{eval}[f]()\\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \}\end{array}}{\forall f.\ \left(\begin{array}{c}\{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \}\\ \lambda f.\ \mathsf{eval}[f]()\\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \}\end{array}\right) \otimes X}\ \mathrm{DFR}}{\forall X.\ \forall f.\ \left(\begin{array}{c}\{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \}\\ \lambda f.\ \mathsf{eval}[f]()\\ \{\ f \mapsto \{\mathsf{emp}\}\ \cdot ()\ \{\mathsf{emp}\}\ \}\end{array}\right) \otimes X}\ \mathrm{Generalisation}
$$

let $hidden$ = new ‘skip’ in

let $runIt$ = new ‘λf . eval$[hidden]()$; $[hidden] := [f]$’ in

  let ctr = new 0 in

  let f_1 = new ‘skip’ in

  let f_2 = new ‘$[ctr] := [ctr] + 1$’ in

    eval $[runIt](f_2)$;

    $[ctr] := 0$;

    eval $[runIt](f_1)$;

    if $[ctr] \neq 0$ then abort else skip

$$e ::= \quad 0 \mid 1 \mid \ldots \mid e_1 + e_2 \mid \ldots \mid x \mid `\lambda\vec{x}.C'$$

$$\begin{aligned} C ::= \quad & \textsf{let } y = [e] \textsf{ in } C \mid [e_1] := e_2 \mid \textsf{let } x = \textsf{new } \vec{e} \textsf{ in } C \mid \textsf{free } e \\ & \mid \textsf{eval } [e](\vec{e}) \\ & \mid \textsf{skip} \mid C_1; C_2 \mid \textsf{if } e_1 = e_2 \textsf{ then } C_1 \textsf{ else } C_2 \end{aligned}$$

Absolutely the end