Lecture 5 notes—Refinement

Butler Lampson
6.826
September 15, 2020

Agenda for today

Review definition of *implements* as subset of traces, with cache example

Too much spec state → history variables, with StatDB example

Abstraction relations ↔ history variables

Premature choice → prophecy variables

Correctness

Definition: External code traces ⊆ external spec traces

Note: this throws away a *lot* of information: internal state and actions

Safety: what code **may** do—never anything bad

Liveness: what code **must** do—eventually something good

A sequential procedure (e.g., $sort(a)$) relates initial and final states.

Correctness: code relation ⊆ spec relation. That is,

Safety: the spec allows the final state of the code (partial correctness)

Liveness: termination (total correctness)

Basic idea for proof: **simulation**—code matches spec one action at a time

Simple example: code for memory with cache

Note: a command may change the state, be non-deterministic. You can only assign its return value. An expression is math: no state change, deterministic.

Spec for memory

type M	$= A \rightarrow V$	also a key-value store
var m	$: \; M$	
$initM()$	$=$ **var** $m' \mid (m'.\text{dom} = A); m := m'$	m' arbitrary, defined
$read(a){:}V$	$= m(a)$	
$write(a, v)$	$= m(a) := v$	$m := m(a := v)$

Writeback cache code

```
type  $C$      $= A \to \mathbf{opt}\, V$                        $c$ defined at *some* $A$'s
const $cSize = 4096$                                  in fact, 4096 of them
var   $mc$     : $M$                                  memory in the code
      $\boldsymbol{c}$      : $\boldsymbol{C}$

$initC()$      $= \mathbf{var}\ c' \mid\ c'.\text{dom.size} = cSize$          $c'$ defined at $cSize$ $A$'s
                        $\wedge\ \forall a \in c'.\text{dom} \mid c'(a) = mc(a)$;
                   $c := c'$

$read(a){:}V$  $= load(a);\ \mathbf{ret}\ c(a)$
$write(a, v)$  $= load(a);\ c(a) := v$
```

Internal

```
$read(a){:}V$    $= load(a);$ ret $c(a)$
$write(a, v)$  $= load(a); c(a) \coloneqq v$

$load(a)$   $=$ if $c(a) \neq None$ then skip                        if $a$ isn't in the cache
                else $\{flush1(); c(a) \coloneqq mc(a)\}$          make space, put it there

$flush1()$  $=$ var $a \in c$.dom;                                  pick an $a$ in the cache
                if $c(a) = mc(a)$ then skip
                else $\{mc(a) \coloneqq c(a)\}$;                       write $a$ to $m$ if dirty
                $c(a) \coloneqq None$                                 take $a$ out of cache
```

Note: the spec is deterministic, but the code is not.

Abstraction function

It has to take the code state (m, c) to the spec state m.

$m \qquad\qquad = mc + c$ function overlay

Writing this out

$m \qquad\qquad = \textbf{fun}\ a \Longrightarrow \textbf{if}\ c(a) \neq None\ \textbf{then}\ c(a)\ \textbf{else}\ mc(a)$

Why code to spec? Because code has many ways to represent the spec
as in the cache example.
(Not always true, though. Sometimes the spec is better with more state.)

An abstraction function F must satisfy two conditions.

Initial: If t is an initial state of T, then $F(t)$ is an initial state of S.
Next (simulation): If t is a reachable state of T
and $t \overset{\pi}{\Rightarrow} t'$ is a step of T,
then there is a step $F(t) \overset{\pi}{\Rightarrow} F(t')$ of S
that has the same trace π.

Simulation

If t is a reachable state of T
and $t \overset{\pi}{\Rightarrow} t'$ is a step of T,
then there is a step $F(t) \overset{\pi}{\Rightarrow} F(t')$ of S
that has the same trace π.

$$\begin{tikzcd}
F(t) \arrow[r, Rightarrow, "\pi"] & F(t') \\
t \arrow[u, "F"] \arrow[r, Rightarrow, "\pi"] & t' \arrow[u, "F"]
\end{tikzcd}$$

The diagram commutes

Theorem: If there is an abstraction function from T to S,
then T implements S, i.e., every trace of T is a trace of S.

Why: Induction on the length of the trace.

Invariant

Invariant: a property that is true of all reachable states.

Must show that code simulates spec from every **reachable** state
The invariant bounds the reachable states.
 Usually the code doesn't simulate the spec from *every* state.

Usually prove P is an invariant by induction on the length of executions.
 P is true initially, and if $P(s)$ and $s \rightarrow s'$ then $P(s')$

The only invariant for the WB cache is the cache size: c.dom.size $= cSize$

Hash table code for the memory spec would have a more interesting one:
 If v is in the bucket with hash h, then $hash(v) = h$.

If cache entries have dirty bits the invariant is: $c(a) \neq m(a) \Rightarrow dirty(a)$

Cache code with dirty

```
type C        = $A \to$ opt $(val{:}\,V, dirty{:}\,\mathrm{Bool})$              $c$ defined at *some* $A$'s
var  $mc$     :  $M$                                              memory in the code
     $c$      :  $C$

$initC()$     = var $c'$ |   $c'.\mathrm{dom.size} = cSize$              $c'$ defined at $cSize$ $A$'s
                         $\wedge\, \forall a \in c'.\mathrm{dom} \mid c'(a) = (mc(a), \mathrm{false})$;
                $c := c'$;

$read(a){:}\,V$  = $load(a)$; ret $c(a).val$
$write(a, v)$ = $load(a)$; $c(a) := (v, \mathrm{true})$
```

***Internal with** dirty*

```
var $mc$     : $M$                                    memory in the code
    $c$      : $C$
$read(a){:}V$ = $load(a)$; ret $c(a).val$
$write(a,v)$ = $load(a)$; $c(a) := (v, \text{true})$

$load(a)$    = if $c(a) \neq None$ then skip          if $a$ is in the cache, done
               else {$flush1()$;                      if not, make space,
                     $c(a) := (mc(a), \text{false})$} and put it there

$flush1()$   = var $a \in c$.dom;                     pick an $a$ in the cache
               if $c(a).dirty$                        write $a$ to $m$ if dirty
               then {$mc(a) := c(a).val$} else skip;
               $c(a) := None$                         and take $a$ out of cache
```

Completeness

The simple abstraction function method always works if the spec avoids

- extra state,
- too few or too many transitions, and
- premature choices.

Why bother? Just don’t make bad choices in the spec?

No. A spec should be written to be as clear as possible to the clients, not to make it easy to prove the code correct.

Extra spec state

Notation: If q: **seq** T and $\oplus$ is an operator on T, $fold(\oplus, q)$ is

$$q_0 \oplus q_1 \oplus \ldots \oplus q_n$$

It works for sets too if $\oplus$ is commutative.

$sum(q\colon \textbf{seq}\ \text{Int}) \qquad = fold(+, q)$

$StatDB$. Definitions (from a textbook):

$$mean = \frac{\sum_i db(i)}{n},\ variance = \frac{\sum_i (db(i) - mean)^2}{n} = \frac{\sum_i db(i)^2}{n} - mean^2$$

```
var db          :  seq V := [ ]                 a multiset

add(v)          = db := db ++ [v]
size(): Nat     = db.size
mean(): opt V   = if db = [ ] then None else sum(db)/db.size
var(): opt V    = if db = [ ]then None
                  else sum({v ∈ db | | (v − mean())²})/db.size
```

Efficient code for StatDB

```
var $count$      $:= 0$
    $sum$        $:= 0$
    $sumSq$      $:= 0$

$add(v)$          $= count := count + 1;\ sum := sum + v;$
                    $sumSq := sumSq + v^2$
$mean()$: opt $V$ = if $count = 0$ then $None$ else $sum/count$
$var()$  : opt $V$ = if $count = 0$ then $None$
                    else $sumSq/count - mean()^2$
```

But there’s no AF, because the spec has more state than the code—there’s no way we can conjure up all of db from the three Nats in the code.

This is not a bad spec; the job of the spec is to be clear, not to be efficient.

History variables

To get an AF, we must add **history variables** to track the extra spec state.

In this case, we just add db, the entire state of the spec, and the spec's action $db \coloneqq db \mathbin{+\!+} [v]$ in add.
Adding all the spec state always works, but often you can add less.

The history variables are not allowed to affect the ordinary variables, so it's obvious that the code with history variables has the *same* traces.

So instead of $code \subseteq spec$, we have $code = code + history \subseteq spec$.

The AF proof needs an invariant that relates db to the rest of the state.

$$\begin{aligned} &/\backslash\ count &&= db.size \\ &/\backslash\ sum &&= fold(+, db) \\ &/\backslash\ sumSq &&= fold(+, db \circ square) \end{aligned}$$

Abstraction relations

Another way: generalize AFs to *abstraction relations*. $\approx$ is an AR if:

Initial: If t is an initial state of T, S has an initial state s such that $t \approx s$.

Next (simulation): If t and s are reachable, $t \approx s$ and $t \overset{\pi}{\Rightarrow} t'$ is a T step, then there exists an s' such that $s \overset{\pi}{\Rightarrow} s'$ is an S step, and $t' \approx s'$.

$$\begin{tikzcd}
s_2 \arrow[rr, Rightarrow, "\pi"] \arrow[dd, leftrightarrow, "\approx"'] & & s_2' \arrow[dd, leftrightarrow, "\approx"'] & \\
& s_1 \arrow[rr, Rightarrow, "\pi"] \arrow[dl, leftrightarrow, "\approx"] & & s_1' \arrow[dl, leftrightarrow, "\approx"] \\
t \arrow[rr, Rightarrow, "\pi"] & & t' &
\end{tikzcd}$$

T and S run in parallel. T drives the execution, doing what it wants.
S follows along, producing the same external trace.
The two conditions guarantee that S can always do this.

AR for StatDB

Here's the AR for $StatDB$

$$\begin{aligned}
&/\backslash\ count &&= db.size \\
&/\backslash\ sum &&= fold(+, db) \\
&/\backslash\ sumSq &&= fold(+, db \circ square)
\end{aligned}$$

It's just the same as the invariant we had for the history variables.

Taking several steps in the spec

If t and s are reachable states of T and S with $t \approx s$, and
$t \overset{\pi}{\Rightarrow} t'$is a step of T, then
there's an *execution fragment* (sequence of steps) of S from s to some s'
with the same trace (one π step, others internal) , and
with $t' \approx s'$.

$$s_0 \Rightarrow \cdots \Rightarrow s_i \overset{\pi}{\Rightarrow} s_{i+1} \Rightarrow \cdots \Rightarrow s'$$

Usually there are many internal code steps for each spec step, e.g., machine instructions.

Sometimes you want internal spec steps, such as the $drop$ action in the async messaging spec.

Premature choice and prophecy variables

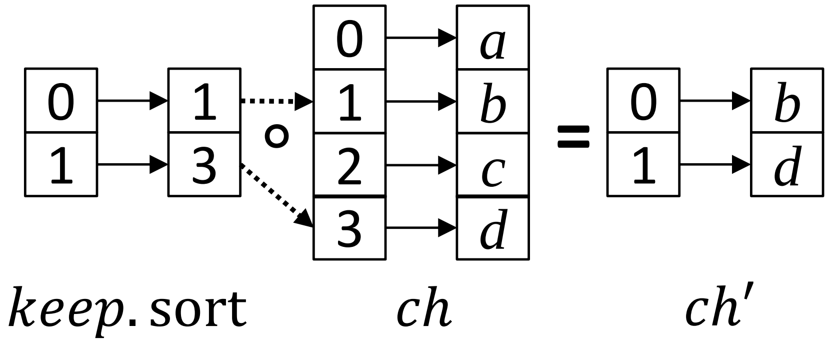
Two realistic examples of premature choice in specs.

Reliable two-party channel spec

var ch : **seq** Msg channel

$send(msg) = ch := ch +\!+ [msg]$
$rcv(addr) =$ **if** $msg = ch.head$ **then** $\{ch := ch.tail;$ **ret** $msg\}$

At a crash, maybe drop some messages.
$crash =$ **var** $keep \subseteq ch.\mathrm{dom}; ch := keep.\mathrm{sort} \circ ch$



Code uses async messaging

var ch : **set** *AsyncMsg*		channel
$send(asm)$	$= ch := ch \cup \{asm\}$	
$rcv()$	$=$ **var** $asm \in ch$ | **ret** asm	doesn’t drop from ch

The drop action is internal, and non-deterministic.

$drop()$	$=$ **var** $asm \in ch;\ ch := ch - \{asm\}$	can drop any time

In real code (such as TCP) a message can be lost long after a crash.

- An asm for a message sent before a crash can stay in the network.
- If the asm is dropped, so is the msg, since there’s no retransmission.
- If the asm is received, so is the msg.

Spec without premature choice

To fix this, mark in-flight messages:

type $MK = (m: Msg, mark: Bool)$ marked message

var ch : **seq** MK channel

$send(msg) = ch := ch \mathbin{++} [(msg, \text{false})]$

$rcv(addr) = \textbf{if}\ mk = ch.head\ \textbf{then}\ \{ch := ch.tail;\ \textbf{ret}\ mk.m\}$

$drop = \textbf{if}\ mk = ch.head \wedge mk.mark\ \textbf{then}\ ch := ch.tail$

$crash = ch := ch \circ \left(\textbf{fun}\ mk \Rightarrow (mk.m, \text{true})\right)$ mark all mk's in flight

Consensus spec

```
**var** $agreed$        :  $\textbf{opt}\,V := None$                          data value to agree on

$allow(v)$              $= \textbf{var}\ a \in \{v, None\};$                      optionally accept $v$
                           $\textbf{if}\ agreed = None\ \textbf{then}\ agreed := a$
$result()$: $\textbf{opt}\,V = \textbf{var}\ a \in \{agreed, None\};\ \textbf{ret}\ a$      optionally return $agreed$
```

This spec chooses the value to agree on as soon as the value is allowed.

$result$ may return $None$ even after the choice is made because in distributed code maybe not all the participants know what the outcome is.

Consensus without premature choice

Code saves allowed values, and the processes chat to choose a value.
The following spec has that form. It has a new $allow$, and adds $agree$

```
var $agreed$  : opt $V$ := $None$                          data value to agree on
var $allowed$ : set $V$ := $\{\}$                          allowed values

$allow(v)$           = $allowed := allowed \cup \{v\}$                      remember $v$
$result()$: opt $V$  = var $a \in \{agreed, None\}$; ret $a$               optionally return $agreed$

$agree()$            = var $a \in allowed$;                                 second half of spec *allow*
                       if $agreed = None$ then $agreed := a$
```

Note: if $result$ couldn't return $None$ even after $agree$, these specs would be different, because the second would allow the behavior

$$allow(1);\ result() = None;\ allow(2);\ result() = 1$$

and the first would not.

Prophecy variables

Abstraction functions can still do the job here, but we need an auxiliary variable that looks into the future, as a history variable looks into the past.

A system TP (T with Prophecy) with a *prophecy variable* has the same traces as the original system T.

Rules for prophecy variables

History variable h	**Prophecy variable p**
1. Every *initial* state has at least one value for h.	1. *Every* state has at least one value for p.
2. No existing step is disabled by new guards involving h.	2. No existing step is disabled *in the backward direction* by p: If $t \overset{\pi}{\Rightarrow} t'$is a step and (t', p') a state, there's a p and a step of TP $(t, p) \overset{\pi}{\Rightarrow} (t', p')$
3. Assigning to a vanilla variable (including a return value) doesn't depend on h.	3. Same: p *can* affect what actions are enabled, but not how an action changes a vanilla variable.
	4. If t is an initial state of T and (t, p) is a state of TP, it's initial.

How to write premature-choice specs

Most people find prophecy variables hard to grasp.

If you have a spec S that makes a premature choice (not likely), deal with it at the highest possible level, as above:

- write another spec S' without premature choice, and
- use a prophecy variable (or sheer willpower) to show that $S' = S$.

Then proving lower level code won’t involve prophecy variables.

Digression on naming actions

There are (at least) two ways to identify the actions.

- The way the diagram and the code names an action by an identifier such as $read$, together with the values of all the arguments and results.

  Thus $read(a){:}\,3$ or $read(a, 3)$.
- The other way only talks about state, and it makes the calling sequence explicit with state variables for the arguments and results.

  So memory has variables $doRead, doWrite, a, v$.

  The action named $read(b, u)$ corresponds to the steps

  $a \coloneqq b, doRead \coloneqq true$;

  $v \coloneqq u, doRead \coloneqq false$

  That is, memory sets $doRead \coloneqq false$ when the result is ready.