Notes for 6.826 lecture 6—Refinement

Agenda for today:
- Review definition of implements, with cache example
- Too much spec state → History variables, with StatDB example
- Abstraction relations ↔ history variables
- Premature choice → prophecy variables

Specs for sequential procedures: code relation <= spec relation. That is, the final state of the code is allowed by the spec (partial correctness) + termination (total correctness)

Definition: External code traces <= external spec traces

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

  Safety (never anything bad) and liveness (eventually something good)

Basic idea: simulation, showing that code matches spec one action at a time

Simple example: memory with cache code.

Note: a command (that changes the state or is non-deterministic) that returns a value can only appear by itself on the right hand side of an assignment; the meaning is that the command does its thing, and in addition the value of the left hand side is changed. An expression is a mathematical expression, that is, it doesn't change the state and it's deterministic.

```
**type** $M$        $= A \to V$                                              also called a key-value store
**var** $m$         $: \; M \coloneqq initM()$

$initM(): M$        $=$ **var** $m' \mid \forall\, a, m'(a) \neq None;$ **ret** $m'$      $m'$ arbitrary, defined everywhere
$read(a): V$        $= m(a)$
$write(a, v)$       $= m(a) \coloneqq v$

**type** $C$        $= A \to$ **option** $V$                                 $c$ is defined at only a few $A$'s
**var** $mc$        $: \; M \coloneqq initM()$                               memory in the code
    $c$             $: \; C \coloneqq initC()$

$initC(): C$        $=$ **var** $c \mid \{a \mid c'(a) \neq None\}.size = cSize;$   $c'$ arbitrary, defined at $cSize$ $A$'s
                      **ret** $c'$

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

**Internal**
$load(a)$           $=$ **if** $c(a) = None$                                 if $a$ isn't in the cache
                      **then** $\{flushOne(); c(a) \coloneqq mc(a)\}$         make space for it and put it there
$flushOne()$        $=$ **var** $a \mid c(a) \neq None;$                     pick an $a$ in the cache
                      **if** $c(a) \neq mc(a)$ **then** $mc(a) \coloneqq c(a);$   write it back to $m$ if necessary
                      $c(a) \coloneqq None$                                  and remove it from the 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.

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

Writing this out

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

Why code to spec? Because code has many ways to represent the spec in general, as in the cache example.

This isn't always true, though. Sometimes it's clearer to write the spec with more state.

An abstraction function F is required to satisfy the following two conditions.

1. If t is any initial state of T, then $F(t)$ is an initial state of S.
2. (Simulation) If t is a reachable state of T and (t, p, t') is a step of T, then there is a step of S from $F(t)$ to $F(t')$ that has the same trace.

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

The diagram commutes

Theorem 1: 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.

Digression on naming actions

There are (at least) two ways to identify the actions. The way in the diagram and in the code we have written names an action by an identifier for the action's code, such as $read$, and the values of all the arguments and results. Thus $read(a) \to 3$ or $read(a, 3)$.

The other way only talks about state, and it makes the calling sequence of the code explicit with state variables for the arguments and results. So memory has variables $doRead, doWrite, a, v$, and the action named $read(b, u)$ corresponds to the steps $a \coloneqq b, doRead \coloneqq true$ and $v \coloneqq u, doRead \coloneqq false$ (that is, memory sets $doRead \coloneqq false$ when the result is ready.

An **invariant** of a module is any property that is true of all *reachable* states of the module.

To use the abstraction function scheme, we must show that the code simulates the spec from every reachable state, and the invariant precisely characterizes the reachable states. It usually isn't true that the code simulates the spec from every state.

The usual way to prove that a property P is an invariant is by induction on the length of finite executions.

There is no invariant for the WB cache. But hash table code for the memory spec would have one, saying that if v is in the bucket with hash h, then $hash(v) = h$.

Completeness and doctrine for specs: If we avoid extra state, too few or too many transitions, and premature choices in the spec, the simple abstraction function method will always work. You might therefore think that all these problems are not worth solving, because it sounds as though they are caused by bad choices in the way the spec is written. But this is wrong. A spec should be written to be as clear as possible to the clients, not to make it easy to prove the correctness of code for.

History variables

Notation: If q is **seq** T or **set** $T$and op is an operator or function that takes two T's, $(op\ :\ q)$ is $StatDB$. 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; don't care about the order

$add(v)$            $= db := db ++ \{v\}$
$size(): Nat$       $= db.size$
$mean()$: option $V$= if $db = \{\}$ then $None$ else $sum(db)$
$vari()$ : option $V$ = if $db = \{\}$ then $None$ else $\frac{sum(\{v \in db \mid\mid (v - mea\ ())^2\})}{db.size}$
```

And efficient code

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

$add(v)$            $= count := count + 1; sum := sum + v; sumSq := sumSq + v^2$
$mean()$: option $V$= if $count = 0$ then $None$ else $sum/count$
$vari()$ : option $V$ = if $count = 0$ then $None$ else $\frac{sumSq}{count} - mean()^2$
```

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 sign of a bad spec; the job of the spec is to be clear, not to be efficient. To get an AF, we must add **history variables**: variables that are added to the state of the code *T* in order to keep track of the extra information in the spec *S* that was left out of the code.

In this case, we just add db, the entire state of the spec. That 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 and without have the same traces.

So instead of $code \rightarrow spec$, we have $code \rightarrow code + history \rightarrow spec$. With the history, we can do an AF proof that $code + history \rightarrow spec$, and it should be obvious that $code \rightarrow code + history$, because the history is not allowed to affect the steps or the ordinary variables.

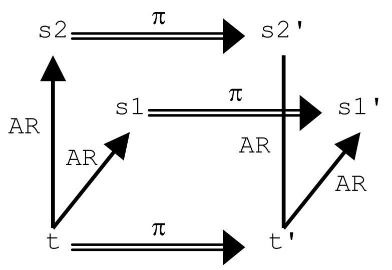
To do the AF proof for the efficient $statDB$ code we need an invariant that relates db to the rest of the state.

$$\begin{aligned}&/\backslash\ count = db.size\\&/\backslash\ sum = (+ : db)\\&/\backslash\ sumSq = (+ : \{v \in db \mid\mid v^2\})\end{aligned}$$

Abstraction relations

Another way to do the same thing is to generalize AFs to abstraction relations. R is an AR if

1. If t is any initial state of T, then there is an initial state s of S such that $(t, s) \in R$.
2. (Simulation) If t and s are reachable states of T and S with $(t, s) \in R$ and (t, π, t') is a step of T, then there is a step of S from s to some s' that has the same trace, and $(t', s') \in R$.



It's sufficient to have a relation. A way to think of this is that the two modules, T and S, are running in parallel. The execution is driven by module T, which executes in any arbitrary way. S follows along, producing the same externally visible behavior. The two conditions above guarantee that there is always some way for S to do this.

Here's the AR for $StatDB$

$$\begin{aligned}&/\backslash\ count = db.size\\&/\backslash\ sum = (+ : db)\\&/\backslash\ sumSq = (+ : \{v \in db \mid\mid v^2\})\end{aligned}$$

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 respectively, with $(t, s) \in R$, and (t, π, t') is a step of T, then there is an *execution fragment* of S from s to some s', having the same trace, and with $(t', s') \in R$.

Usually there are many internal code steps for each spec step, e.g., machine instructions. Sometimes it's convenient to have extra internal spec steps, such as the $Drop$ action in the async messaging spec.

Premature choice and prophecy variables

Some realistic examples of premature choice in specs.

Reliable two-party channel spec

```
var  ch            : seq Msg                                   channel

send(msg)          = $ch := ch ++ \{msg\}$
receive(addr)      = $msg = ch.head \rightarrow ch := ch.tail$; ret $msg$

crash              = var $keep \subseteq ch.dom$; $ch := keep.sort \circ ch$
```

Most practical code (for instance, the Internet's TCP protocol) has cases in which it isn't known whether a message will be lost until long after the crash. This is because they ensure FIFO delivery, and get rid of retransmitted duplicates, by numbering messages sequentially and discarding any received message with an earlier sequence number than the largest one already received. If the underlying message transport is not FIFO (like the Internet) and there are two undelivered messages outstanding (which can happen after a crash), the earlier one will be lost if and only if the later one overtakes it. You don't know until the overtaking happens whether the first message will be lost. By this time the crash and subsequent recovery may be long since over.

To fix this, mark in-flight messages:

```
var  ch            : seq (Msg, Bool)                           channel

send(msg)          = $ch := ch ++ \{(msg, false)\}$
receive(addr)      = $t = ch.head \rightarrow ch := ch.tail$; ret $t.msg$
drop               = $t = ch.head \wedge t.mark \rightarrow ch := ch.tail$

crash              = $ch := map((\textbf{fun}\ (m, b) \rightarrow (Msg, Bool) \Rightarrow (m, true)), ch)$
```

Consensus spec

```
var  outcome       : option $V := None$                        data value to agree on
allow(v)           = var $o \in \{v, None\}$;                  optionally accept v
                       if $outcome = None$ then $outcome := o$ if $outcome$ is $None$
result():option V  = var $o \in \{outcome, None\}$; $o$         optionally return $outcome$
```

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 it's possible that not all the participants have heard what the outcome is.

Code for almost certainly saves up the allowed values and does a lot of communication among the processes to come to an agreement. The following spec has that form. It includes the spec above except for $allow$, and adds

```
var allowed        :  set $V := \{\}$                           data value to agree on
allow(v)           = $allowed := allowed + \{v\}$               remember $v$
agree()            = var $o \in allowed \cup \{None\}$; if $outcome = None$ then $outcome := o$
```

Note that if $result$ didn't have the option to return $None$ even after $outcome \neq None$, these specs would not be equivalent, because the second would allow the behavior

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

and the first would not.

Our trusty method of abstraction functions can still do the job here. However, we have to use a different sort of auxiliary variable, one that can look into the future just as a history variable looks into the past. Just as we did with history variables, we will show that a module TP (T with Prophecy) augmented with a *prophecy variable* has the same traces as the original module T. Actually, we can show that it has the same *finite* traces, which is enough to take care of safety properties.

History variable	*Prophecy variable*
1. Every initial state has at least one value for the history variable.	1. *Every state* has at least one value for the prophecy variable.
2. No existing step is disabled by new guards involving a history variable.	2. No existing step is disabled *in the backward direction* by new guards involving a prophecy variable. More precisely, for each step (t, π, t') and state (t', p') there must be a p such that there is a step $((t, p), \pi, (t', p'))$.
3. A value assigned to an existing state component must not depend on the value of a history variable. One important case of this is that a return value must not depend on a history variable.	3. Same condition. A prophecy variable *can* affect what actions are enabled, subject to condition (2), but it can't affect how an action changes an existing state component.
	4. If t is an initial state of T and (t, p) is a state of TP, it must be an initial state.

Most people find this hard to grasp. In the unlikely event that you have to deal with a spec that makes a premature choice, you should deal with it at the highest possible level, as in the two examples above. That is, write another spec that does not make a premature choice, and use a prophecy variable (or sheer willpower) to show that it implements the first one. Then when you deal with more realistic code, you won't have to worry about prophecy variables.