6.894 Lightweight formal methods

Lecture 15: Weakest preconditions

Daniel Jackson, April 13, 2005.

Topics:

- Dijkstra's Guarded Commands
- Weakest preconditions
- ESC / Java.

Why Guarded Commands?

A 'kernel' programming language for research in program verification.

Now used as an intermediate language by ESC/Java and other tools that generate verification conditions.

Syntax of (Guarded) Commands

Replace conditionals and loops:

if	do
$B_1 \to S_1$	$B_1 \to S_1$
$\square\ B_2 \to S_2$	$\square\ B_2 \to S_2$
⋮	⋮
$\square\ B_n \to S_n$	$\square\ B_n \to S_n$
fi	od

$B_1 \ldots B_n$ are guards: boolean expressions
$S_1 \ldots S_n$ are commands

Guards of an if-fi or do-od can overlap.
Semantics of if-fi: pick any guard that's true and execute its command.

Semantics of do-od:

Pick a guard that is true and execute its command, repeatedly until no guard is true.

(Can actually generalize this language, and make $\square, \rightarrow$ operators, so that syntax of commands becomes

$S ::= B \rightarrow S \mid S_1 \square S_2 \mid$ do S od $\mid$ if S fi

See Greg Nelson: A Generalization of Dijkstra's Calculus.)

Why is this useful? (from Apt & Olderog)

1. Symmetry

Compare versions of GCD algorithm:

```
while x ≠ y do
  if x > y then x := x - y else y := y - x
```

```
do
  x > y → x := x - y
□ x < y → y := y - x
od
```

2. Failures

if $B_1 \rightarrow S_1$ fi fails if ~~none of~~ ~~B1~~ ~~true~~. B false.

Can use to make array bounds etc:

if $0 \leq i < n \rightarrow x := a[i]$ fi

Non-determinism.

Sometimes don't need determinism even in seq. program:

```
twop := 0 ; threep := 0
do 2 divides x → x := x div 2 ; twop++ ;
   3 divides x → x := x div 3 ; threep++ ;
od
```

if 6 divides x, either can be chosen.

Modelling concurrency.

Can model || as interleaving

$$S \equiv [x := 0 \;\|\; x := 1 \;\|\; x := 2]$$

```
turn1 := true ; turn2 := true ; turn3 := true ;
do
   turn1 → x := 0 ; turn1 := false ;
   ⋮
od
```

Hoare Logic for GCs

$$\frac{\{p \wedge B_i\}\ S_i\ \{Q\} \qquad i \in 1..n}{\{P\}\ \underline{\text{if}}\ \Box_{i=1}^{n} B_i \to S_i\ \underline{\text{fi}}\ \{Q\}}$$

$$\frac{\{p \wedge B_i\}\ S_i\ \{p\}}{\{P\}\ \underline{\text{do}}\ \Box_{i=1}^{n} B_i \to S_i\ \underline{\text{od}}\ \{p \wedge \bigwedge_{i=1}^{n} \neg B_i\}}$$

Weakest preconditions

A problem with HL: always reasoning backwards from post condition, but logic doesn't reflect this.

(Why reason backwards? Could start at precondition and move forward, generating strongest postcondition and seeing if it implies the post condition we need. But this wastes work: better to start with the given post-condition, and move backwards to a weakest precondition and see if precond implies it.

Question: but doesn't this waste effort too?
Yes, if precond is very strong. But that's rare; usually have weak pre and strong post.
)

wp (S, R) — weakest precondition of stmt S with respect to post condition R.

Liberal

wlp (S, R) — weakest ~~[illegible]~~ precondition.

wp(S,R) is a predicate defining the largest set of states from which exec of S is guaranteed to terminate in a state satisfying R.

wlp (S,R) is the same but for partial correctness: set of states that lead to states in R, or non-termination (or failure).

Relationship between wp and wlp.

$$\underbrace{wp(S;R)}_{\text{total correctness}} = \underbrace{wp(S, TRUE)}_{\text{termination}} \wedge \underbrace{wlp(S,R)}_{\text{partial correctness}}$$

Predicate transformers.

You can view $wp(S;R)$ as $wp_S(R)$, where wp_S is a predicate transformer, from (postcondition) predicates to (precondition) predicates.

[Note: in class, I claimed wrongly that the backwards step in model checking of EF p is a wp step. Of course it isn't: the step corresponding to EX p (iterated to a fixpoint for EF p) is backwards relational image, giving the set

$$P' = \{s \mid \exists s' \in P \,.\, (s,s') \in T\}$$

whereas the weakest precondition is

$$P' = \{s \mid \forall s' \,.\, (s,s') \in T \Rightarrow s' \in P\}$$

which corresponds instead to AX P, not EX P.
]

Rules for obtaining weakest preconditions

We'll focus on wlp, for partial correctness

$wlp\ (SKIP, R) = R$

$wlp\ (A; B, R) = wlp\ (A, wlp\ (B, R))$

$wlp\ (x := e, R) = R[x := e]$

$R[x := e]$ is R with e substituted for x — that is, every occurrence of x replaced by e. Also written $R[e/x]$, $R[x \leftarrow e]$

$wlp\ (\text{if } [] B_i \rightarrow S_i \text{ fi}, R) = \bigwedge_i B_i \Rightarrow wp\ (S_i, R)$

Wp of Loops

Weakest pre of a loop is a fixpoint, and can't be expressed as a syntactic construction in terms of the post condition.

In practice, find some precondition of the loop, usually not the weakest

Theorem.

if $P \wedge B_i \Rightarrow wp(S_i, P) \quad \forall i : 1..n$

then

$$P \Rightarrow wp(\mathbf{do}\ \Box_{i=1}^{n} B_i \rightarrow S_i\ \mathbf{od}, P \wedge \neg \wedge_i \neg B_i)$$

Note.

$$P \rightarrow wlp(S, R) \equiv \{P\}\ S\ \{R\},$$

so this is exactly the same as the Hoare loop rule.

Examples, Weakest Preconditions

① Prove correctness of

SWAP ≙ $t := x \; ; \; x := y \; ; \; y := t$

② Prove correctness of:

MAX ≙ **if**

$x > y \longrightarrow m := x;$

$y > x \longrightarrow m := y;$

$x = y \longrightarrow m := x;$

fi

for postcondition $\{(m = x \vee m = y) \wedge x \leq m \wedge y \leq m\}$

③ Prove correctness of

SUM ≙ $s := 0;$

$i := 0$

do

$i < n \longrightarrow s := s + a[i]; \; i := i + 1;$

od

Examples : solutions

① SWAP.

post condition is $x = Y \wedge y = X$

where X, Y are initial values of x and y.

$wlp(t := x ; x := y ; y := t, \quad x = Y \wedge y = X)$

$= wlp(t := x, \; wlp(x := y ; y := t, \quad x = Y \wedge y = X))$

[by composition rule]

$= wlp(t := x, \; wlp(x := y, \; wlp(y := t, \quad x = Y \wedge y = X)))$

[by composition rule, again]

$= wlp(t := x, \; wlp(x := y, \quad x = Y \wedge t = X))$

[by assignment rule]

$= wlp(t := x, \quad y = Y \wedge t = X)$

[by assignment rule, again]

$= y = Y \wedge x = X$, as expected

(2) MAX.

$wlp(MAX, (m=x \vee m=y) \wedge x \leq m \wedge y \leq m)$

$= \quad x > y \Rightarrow (x=x \vee x=y) \wedge x \leq x \wedge y \leq x$
$\wedge \quad y > x \Rightarrow (y=x \vee y=y) \wedge x \leq y \wedge y \leq y$
$\wedge \quad x = y \Rightarrow (x=x \vee x=y) \wedge x \leq x \wedge y \leq x$

[by rule for alternative command if fi]

$= \quad x > y \Rightarrow y \leq x$
$\wedge \quad y > x \Rightarrow x \leq y$
$\wedge \quad x = y \Rightarrow y \leq x$

[by properties of equality]

$=$ true

[by properties of $>, \leq$]

(3) SUM -

postcondition is

$$s = \sum_{j=0}^{n-1} a[j]$$

using the strategy "replace constant by variable" suggests the loop invariant

$$INV \triangleq s = \sum_{j=0}^{i-1} a[j]$$

Let's try and check that it's a loop invariant by computing

~~wlp (s := s + a[i] , INV)~~

wlp (s := s + a[i] ; i := i+1 , INV)

= wlp (s := s + a[i], wlp (i := i+1 , INV))

= wlp (s := s + a[i], $s = \sum_{j=0}^{i} a[j]$)

= $s + a[i] = \sum_{j=0}^{i} a[j]$

Call this <u>INV'</u>

Now we check the iteration theorem's condition

$$i<n \wedge INV \Rightarrow INV'$$

$$\Leftrightarrow \quad i<n \wedge s = \sum_{j=0}^{i-1} a[j] \Rightarrow s + a[i] = \sum_{j=0}^{i} a[j]$$

$$\Leftrightarrow \quad \text{true}$$

Note how the constraint $i<n$ makes $a[i]$ defined; without it, we couldn't establish the equivalence of the two summation formulas.

So now we have that INV is indeed a loop invariant.

To establish the post-condition, we need

$$\neg(i<n) \wedge s = \sum_{j=0}^{i-1} a[j] \Rightarrow s = \sum_{j=0}^{n-1} a[j]$$

Now we see that our loop invariant was NOT strong enough. We need additionally $i \leq n$, (left as exercise* for reader), which will give us

$$\neg(i<n) \wedge i \leq n \wedge s = \sum_{j=0}^{i-1} a[j] \Rightarrow s = \sum_{j=0}^{n-1} a[j]$$

which holds trivially because

$$\neg(i<n) \wedge i \leq n \Rightarrow i = n$$

* no this, and you'll see how the loop condition plays a role.

A Question.

Can you patch additional constraints in like this? That is, if I've ~~shown that~~ computed wlp (S, R), and I realize that instead of R, I really wanted R ∧ EXTRA, can I just compute wlp (S, EXTRA) and add it in?

Answer: Yes, because wlp and wp have some nice properties, including

$$wlp(S, R_1 \wedge R_2) = wlp(S, R_1) \wedge wlp(S, R_2)$$

i.e. wlp distributes over conjunction.

ESC / Java.

Tool developed at SRC in Palo Alto.
(DEC, then Compaq, then HP)

Based on work of Greg Nelson primarily;
recent work on extensions to handle encapsulation
due primarily to Rustan Leino (now at Microsoft).
Leino's latest project Spec# is essentially ESC/C#.

Idea.

Use up technology to discharge proof obligations
from assertions
either written by programmer
or implicit because of
* array bounds violations
* null pointer derefs

Engineering + research challenges.

Architecture for generating verification conditions:
> translate to Dijkstra's GC
> pass VC's to specialized theorem prover
Doing proofs automatically
> Simplify, state of the art thm. prover
> built-in decision procedures.
Specification language
> Originally had their own
> Now JML: Java Modeling Language

Loop invariants.

Punt on it!

Unroll loop once.

Generate GC saying that all unrollings that are longer lead to successful states.

Encapsulation, etc.

Lots of subtle research + clever solutions

~~Extend~~

Exceptions : Extend outcome to include these.

Extensions to Dijkstra's basic GC language

① pre and post conditions for method specs.

"requires"

"ensures"

② assert P

fails unless P holds at this point

③ assume P

the statement that magically causes P to hold.

~~wlp(assert P, R) = P ∧ R~~

~~wlp(assume P, R) = P ⇒ R~~

Extension for different outcomes

wlp.C (N, X, W)

holds in initial states
from which each execution of C
terminates normally in state satisfying N
exceptionally X
erroneously W.

wlp.(v = e) (N, X, W) = N[e/v]
wlp.skip = N
raise = X
assert e = (e ∧ N) ∨ (¬e ∧ W)
assume e = e ⇒ N

Very clever treatment of variable introduction:

wlp(var v in C).(N, X, W) = ∀v. wlp.C (N, X, W).

Verification condition for whole method is:
BP ⇒ wlp.C (true, true, false)
T
background predicate:
axioms about Java types etc.

Requires and ensures are desugared into assert + assumes:

```
m(){
    assume requires-m

    n()   assert requires-n
          assume ensures-n      ] a call, replaced by spec of called procedure.

    assert ensures-m
}
```