CSE503: Software Engineering

David Notkin
University of Washington
Department of Computer Science & Engineering
Winter 2002

Weakest preconditions: predicate transformers

- An alternative formulation to Hoare triples is to use Dijkstra's weakest precondition predicate transformers
- In essence, they provide a functional approach to proving programs correct
- **wp(Q,**`S`**)** is defined to be the weakest logical predicate that must be true before executing `S` in order to prove that postcondition **Q** is true afterwards

Examples

- **wp(x > 10,**`x:=x+1`**) = x > 9**
- **wp(Q,**`skip`**) = Q**
- **wp(Q,**`S1;S2`**) = wp(wp(Q,**`S2`**),**`S1`**)**
- **wp(Q,** `if` **C** `then S1 else S2`**) =**
  (wp(Q,`S1`**) and C) or (wp(Q,**`S2`**) and not C) =**
  (C implies wp (`S1`**,Q)) and**
  (not C implies wp(`S2`**,Q))**

wp and Hoare triples

- In general, there is a direct analogy between Hoare triples and wp proofs
- On the whole, wp proofs seem a little more straightforward, as they seem to capture the intuitive notion of backwards reasoning for proofs more effectively

A wp proof example

wp(x > 0, `if x > 0 then x := x + 1 else x := -x fi`**)**

(wp(x > 0,`x := x + 1`**) and x > 0) or**
(wp(x > 0,`x := -x`**) and x <= 0)**

(x > 0 and x + 1 > 0) or (x <= 0 and –x > 0)

(x > 0 and x > -1) or (x <= 0 and x < 0)

x > 0 or x < 0

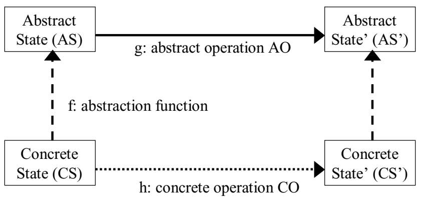
x <> 0

Proofs of Abstract Data Types: related problem to proofs of functions

- Given an abstract data type **A** and an implementation `C` of that type, how do we prove that `C` satisfies **A**'s specification?
- The approach I'll sketch is due to Hoare
  - Precisely define the ADT and the implementation, including invariants over their representations
  - Define an abstraction function that maps a state in the implementation to a state in the abstract representation
  - Very roughly, show for all legal implementation states that applying the abstraction function followed by an abstract operation is equivalent to applying the associated concrete operation followed by the abstraction function

Pictorially:

g(f(CS)) = f(h(CS)) = AS’



7

Precisely define A:

use pre- and post-conditions

```
ADT Stack is
  private s == <>
  const maxSize;
  Push(x) { pre == !Full()
            post == s' = s || <x> }
  Pop()   { pre == !Empty()
            post == s' = <s1..sN-1> }
  Top(): int  { pre == !Empty()
                post == Top = sN and s' = s }
  Empty(): bool  { post == Empty = (s = <>) and
                                    s' = s }
  Full(): bool  { post ==
                  Full = (len(s) = maxSize and s' = s }
end Stack
```

8

Abstract invariants

- We may also define an invariant over the abstraction representations
  - All legal abstract states satisfy these invariants
  - No abstraction operation may map a legal abstract state to one that violates the invariants
- In this case, we define the abstract invariant `len(s) <= maxSize`

9

Implementation

- The implementation must represent the sequence in a concrete way (here, as an array)
-
```
package stackImpl is
  private int sArray[1..maxSize];
  private int sTop := 0;
  procedure sPush(int x) is
    if !sFull() then
      sTop := sTop +1;
      sArray[sTop] := x;
    fi
  end
  procedure…
```

10

Concrete invariants

- We may also define a concrete invariant
- In this case a concrete invariant is `sTop <= maxSize`

11

Status

- In essence, the abstraction specifies how to map an abstract stack to another abstract stack (perhaps with a return value of the operation)
- And similarly, the implementation describes how to map a concrete stack to another concrete stack

12

Abstraction function

- However, there is no association between the abstract type and its implementation
- To associate them we use an explicit abstraction function (AF) that maps a state in the concrete implementation to a state in the abstract representation
  - That is, it describes how to interpret a state in the concrete machine as a state in the abstract machine
  - For the example, the abstraction function maps the `[1..sTop]` elements in `sArray` to a sequence of elements in the stack's `s` variable

13

Abstraction function: many-to-one

- In general (and in this example), many different concrete states map to the same abstract state
- `sTop = 3` and `sArray = [1,2,3,4,5,6]` and `sTop = 3` and `sArray = [1,2,3,6,5,4]` map to the same abstract state: `s = <1,2,3>`
- The many-to-one nature of abstraction functions is one reason that the inverse relation that maps abstract to concrete states is less useful in proofs (although it seems sensible on the face of it)

14

Proof sketch: reprise

- For each initial concrete state `i`, show that `AF(i)` satisfies the abstract invariant
  - That is, make sure that any initial concrete state is a legal abstract state
- For each operation of the concrete implementation and its associated abstract operation, given a legal concrete state `c`, we need to show $\texttt{op}_{abs}\texttt{(AF(c))} = \texttt{AF(op}_{conc}\texttt{(c))}$
- Essentially, this is a proof by induction that shows that the concrete operations satisfy the abstract operations

15

Brief and partial example

- Abstract state: **s = <3,1,4,1,5>**
- Concrete state:
  - `sTop = 5`
  - `sArray = [3,1,4,1,5,17,2,8,2,8]`
- **push(9)**
  - it's !Full()
  - it returns a new **s = <3,1,4,1,5,9>**
- `sPush(9)`
  - `It's also not sFull()and returns`
    - `sTop = 6`
    - `sArray = [3,1,4,1,5,9,2,8,2,8]`

16

Finish example: $\texttt{op}_{abs}\texttt{(AF(c))} = \texttt{AF(op}_{conc}\texttt{(c))}$

- c = `sTop = 5, sArray = [3,1,4,1,5,17,2,8,2,8]`
- **AF(c) = <3,1,4,1,5>**
- **push(9) = <3,1,4,1,5,9>**

- **sPush(9)** = `sTop = 6, sArray = [3,1,4,1,5,9,2,8,2,8]`
- **AF(…) = <3,1,4,1,5,9>**

17

Dealing with "real" language features in proofs

- Side-effects
- Parameters
- Exceptions
- goto's
- OO dispatching
- …

- Most, if not all, of these have been addressed in the literature: as you can imagine, the proof rules and obligations become increasingly complex

18

Proving programs correct: useful for a number of reasons

- Provide a sound basis for understanding the relationship between specifications and implementations
- When working on code, it gives you an intellectual tool (which you can and will and probably should use informally most of the time)
- I don't think about loop invariants in 99% of the loops I write, but when I am having a problem, I often informally write down or at least determine what the loop invariant should be
- When debugging a system that uses ADTs, it may help you separate out potential problems in (a) the abstract definition, (b) the concrete implementation, and (c) the abstraction function between them

Proofs: issues not addressed (reprise)

- Requirements engineering
  - Where did the specification come from? Does it satisfy the needs of the customer?
- Design
  - How does it interact with other parts of the program?
- Evolution
  - What happens if the specification is changed?
- Economics
  - What is the cost of proving correctness?
- Testing
  - Should we rely entirely on the proof?
- …

Next: requirements and specifications