VC's for Functions and Pointers

Klaus v. Gleissenthall

VRIJE
UNIVERSITEIT
AMSTERDAM

Logic:
- Propositional
- First order
- Theories

Programs to Logic:
- Hoare Logic
- VCGen

Automation
- Horn clauses
- Predicate Abstraction

Security
- Information Flow Control
- Side-Channels

Where are we? Recap

- The trickiest case is the one for while loops
- While loops require loop invariants, that is formulas that hold before and after each loop iteration
- Not all invariants work for verification! The proof rule requires an inductive invariant
- Such an invariant allows us to prove that, if the invariant holds *before* the loop iteration it also holds *after* the loop iteration.
- That is, for invariant I, loop condition b and loop body s, we can prove

  $\vdash \{I \wedge b\}\ \mathrm{s}\ \{I\}$
- Finding inductive invariant is the most important problem in deductive verification

Where are we? Recap

- Next, we looked at automating proofs in Hoare logic using weakest preconditions
- **wp**(s, Q) denotes the weakest formula that needs to hold before s,
  to ensure that Q holds *after* s.

- Most rules are straight forward from the Hoare proof rules
- Again, the most difficult part is loops
- The weakest precondition of a loop is its invariant
- But, we still need to check that the invariant is inductive
- This requires side-conditions, which we generate via function vc

Weakest Preconditions: Loops

- What's the weakest precondition for a loop: W $\triangleq$ while b do s
- From our semantics, we know that we can unwind the loop as follows

  if b then s else skip; while b do s

- Then, we can derive
  - wp$(W, Q) = (b \rightarrow$ wp$(W, Q)) \wedge (\neg b \rightarrow Q)$
- But that's a recursive equation, so we're not really any further
- Idea: our Hoare logic proofs used invariants. Let's compute wp wrt. a given invariant

Weakest Preconditions: Loops

- What's the weakest precondition for a loop?
- Say all our loops are annotated with a loop invariant *I*

$$W \triangleq \text{while } [I]\ b \text{ do } s$$

Quiz:

- What should we set wp(W, Q) to?

Weakest Preconditions: Loops

- What's the weakest precondition for a loop?
- Say all our loops are annotated with a loop invariant I

$$W \triangleq \text{while } [I]\ b \text{ do } s$$

Quiz:

- Is it sound to let wp$(W, Q) \triangleq I$?
- No! We need to check that I implies post-condition Q
- We need to check that I is actually an inductive loop invariant!
- Define function vc(s) that encodes these additional conditions

Verification Conditions

- How should we define vc(while $[I]$ b do s, Q)?

- We need to ensure that Q holds *after* the loop, that is $(I \wedge \neg b) \Rightarrow Q$

- I needs to be a loop invariant, that is, needs to be preserved under s, i.e.,

$$\{I \wedge b\}\ \mathrm{s}\ \{I\}$$

- We can prove this by showing $I \wedge b \Rightarrow \mathrm{wp}(s, I) \wedge \mathrm{vc}(\mathrm{s}, I)$

- This means, we can define

$$\mathrm{vc}(\text{while } I\ b \text{ do } s,\ Q) \triangleq (I \wedge b \Rightarrow \mathrm{wp}(s, I)) \wedge \mathrm{vc}(\mathrm{s}, I) \wedge (I \wedge \neg b) \Rightarrow Q$$

Verification Conditions

Quiz: $\mathrm{vc}(\text{while } [I]\ b \text{ do } s, Q) \triangleq (I \land b \Rightarrow \mathrm{wp}(s, I)) \land \mathrm{vc}(s, I) \land (I \land \neg b) \Rightarrow Q$

- Let W $\triangleq$ while $[x \leq 6]$ $x \leq 5$ do $x := x + 1$; we want to prove $\{x \leq 0\}$ W $\{x = 6\}$
- What do we get for $\mathrm{vc}(\mathrm{W}, x = 6)$?

Verification of Hoare Triples

- To show validity of a Hoare triple $\{P\}$ s $\{Q\}$, we thus need to
  - Compute $\mathrm{wp}(s, Q)$
  - Compute $\mathrm{vc}(s, Q)$
- Then $\{P\}$ s $\{Q\}$ is valid, if the following formula is valid

$$\mathrm{vc}(s, Q) \wedge (P \rightarrow \mathrm{wp}(s, Q)) \quad (*)$$

- Thus, if we prove (*), we have shown that the program conforms to its specification

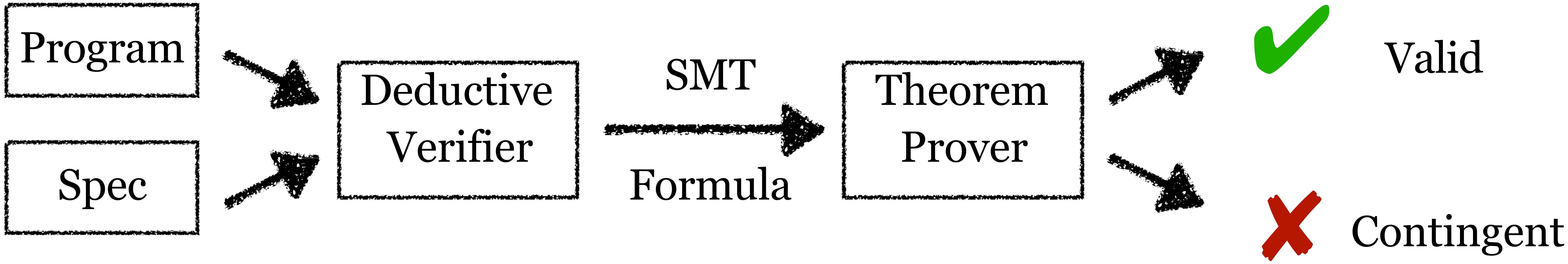
Verification of Hoare Triples

Quiz:

- Is our method complete, that is if $\{P\}$ s $\{Q\}$, then the following formula is valid

$$\mathrm{vc}(s, Q) \wedge (P \rightarrow \mathrm{wp}(s, Q)) \quad (*)$$

Program Verification



- <u>Spec</u>: Assume, Assert, Loop invariants
- <u>Verification condition</u>: An SMT formula ϕ s.t. program is correct iff ϕ is valid

Where are we? Recap

- Now: two important extensions:
  - Functions
  - Pointers
- Next lectures:
  - Horn clauses (good way to represent conditions on loop invariants)
  - Algorithms to find invariants automatically

Extensions & Plan

- Nano is missing many features of real programming languages
- We will now look at two extensions:
    - Functions
    - Pointers
- After that, we'll look at techniques to discover loop invariants (semi-) automatically

Assertions: Syntax

- But first, we will add three new statements to our language
  - The statement **assert**(F) **fails** if F evaluates to $\bot$
  - The statement **assume**(F) **tells us** that F evaluates to $\top$
  - The statement x := **havoc**() assigns a **non-deterministic value** to variable x

Assertions: Semantics

- Add new failure state **fail**, i.e., our state is now either a function σ or **fail**

$$(\text{assert-}\top)\ \frac{\sigma \vDash F}{\langle\, \textbf{assert}(F), \sigma\rangle \Downarrow \sigma} \qquad (\text{assert-}\bot)\ \frac{\sigma \nvDash F}{\langle\, \textbf{assert}(F), \sigma\rangle \Downarrow \textbf{fail}}$$

- Assert behaves like **skip**, in case the assertion holds, and otherwise enters the failure state

$$(\text{assume-}\top)\ \frac{\sigma \vDash F}{\langle\, \textbf{assume}(F), \sigma\rangle \Downarrow \sigma}$$

- Assume only needs a single rule
- If the assumption holds, it behaves like **skip**
- Else, the execution gets "stuck", (but doesn't fail)
    - Thus, when proving partial correctness, we can ignore such executions

Assertions: Semantics

- Finally, x := **havoc**() resets variable x to some non-deterministically chosen value

$$(\text{havoc}) \quad \frac{n \in \mathbb{Z}}{\langle\, x := \textbf{havoc}(),\ \sigma\rangle \Downarrow \sigma[x \mapsto n]}$$

Quiz:

- After including **havoc** in our language, is $\langle s, \sigma\rangle \Downarrow \sigma'$ still a (partial) function?

- **havoc** introduces non-determinism!

Assertions: Proof Rules

- Proof rule for assertions:

$$\frac{P \Rightarrow \mathrm{F}}{\vdash \{P\}\ \mathbf{assert}(\mathrm{F})\ \{P \wedge F\}}$$

- Proof rule for assumption:

$$\frac{}{\vdash \{P\}\ \mathbf{assume}(\mathrm{F})\ \{P \wedge F\}}$$

- Proof rule for havoc:

$$\frac{}{\vdash \{\forall \mathrm{y}.\mathrm{Q}[\mathrm{y}/\mathrm{x}]\}\ \mathrm{x} := \mathbf{havoc}()\ \{Q\}}$$

- If Q holds, no matter what we choose for x, then Q holds after.

Assertions

Quiz:

- What's wp(**assert**$(P), Q)$?

- What's wp(**assume**$(P), Q)$?

- Given statement s, can we transform it into a statement s' such that $\{P\}$ s $\{Q\}$ holds if and only if $\{\top\}$ s' $\{\top\}$ holds?

Functions

- Let's add functions to our language

Program: $P \ni \text{Prog} ::= F^{+}$ (one or more functions)

Function: $F \ni \text{Fun} ::= \textbf{fun}\ f(x_1, \ldots, x_n)\{s;\ \textbf{return}\ e;\}$

Statement: $s \ni \text{Stmt} ::= x := f(e_1, \ldots, e_n) \mid \ldots$

- Aside: we can use the name functions, procedures, method calls
- Often, using procedure or method call is done to indicate that the functions have side-effects

Handling Functions

- How do we generate VCs if we encounter function calls?

$$x := f(e_1, \ldots, e_n)$$

- Just like we asked programmer to provide loop invariants, also ask them for method <u>pre-</u> and <u>post-conditions</u>

- Preconditions specify what is expected of f's arguments

- Postconditions describe f's <u>return value</u> and its possible <u>side-effects</u>

Pre- and post- Example

Quiz:

- Consider a function *get* that takes an array a of size n and index i and returns the i'th element
- What would be a good pre-condition on inputs a,n, and i?
- What would be a good post-condition for return value *ret*?
- Together, pre-, and post-condition are also called <u>function contract</u>

Generating VCs for method calls

- Contracts make verification modular, that is, we can verify one function at a time
- But how can we use a contract for verification?
- There are two questions we need to answer:

    1. How do we verify that a method satisfies its contracts?
    2. How to use the contract when generating VCs for method calls?

1. Verifying Contracts

- Consider the following function declaration:

```
fun f(x₁, ..., xₙ)
  { requires(Pre);
  ensures(Post); s;
  return e;
 }
```

Quiz:

- Let's assume post refers to return value e using the name *ret*
- Which Hoare triple do we have to prove for statement s; ret:=e; ?

2. Verifying Calls

- Which verification conditions should we generate if we encounter a function call?

$$x := f(e_1, \ldots, e_n)$$

- Say our function has arguments $x_1, \ldots, x_n$ and pre-condition *Pre* and post-condition *Post*

Quiz:

- What needs to hold *before* the function call?
- What holds *after* the function call?

2. Verifying Calls

- Which verification conditions should we generate if we encounter a function call?

$$x := f(e_1, \ldots, e_n)$$

- Say our function has arguments $x_1, \ldots, x_n$ and pre-condition *Pre* and post-condition *Post*

- We can replace the function call by the following code, where *tmp* is a fresh variable

$$\textbf{assert}(Pre[e_1/x_1, \ldots, e_n/x_n]);\ \ \textbf{assume}(Post[tmp/ret, e_1/x_1, \ldots, e_n/x_n]);\ \ x := tmp$$

Quiz:

- Why do we need the last assignment?
- Why does *tmp* have to be fresh?

Modular Verification

- When verifying a function definition:
    - We **assume** the precondition
    - And **assert** the postcondition
- When verifying a function call:
    - We **assert** the precondition
    - We **assume** the postcondition
- This is crucial for modular verification –
  decompose verification tasks into individual functions

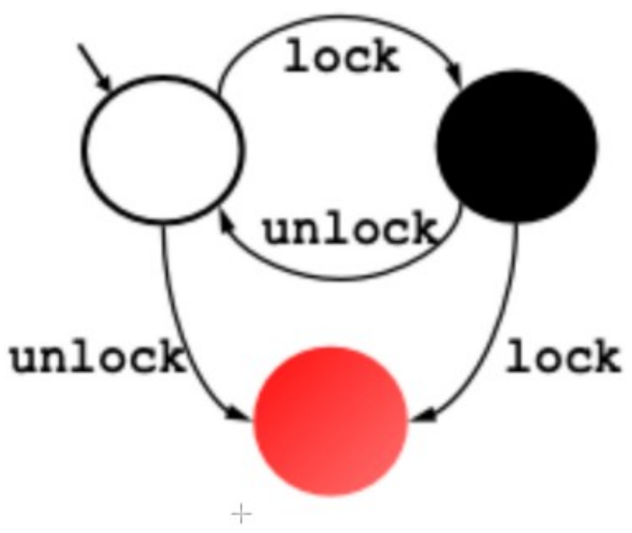
Modular Verification

Quiz:

- Say we don’t have function pre and postconditions
- Is there still some way we could verify programs with functions?
- What’s the downside?
- What’s the downside of modular verification?

Modular Verification

Exercise:



Quiz:

*"An attempt to re-acquire an acquired lock or release a released lock will cause a **deadlock**."*

- Suppose we represent locks as integers – 0 means locked; 1 means unlocked
- What are the contracts for methods lock and unlock?

Modular Verification

Exercise:

- Eliminate the function calls in the following caller of lock and unlock

```
assume(b=0 ∨ b=1);
l:= b;
if(b ≠ 0) l := lock(l); else l := unlock(l);
if(b = 0) l:= lock(l) else l:= unlock(l);
```

- Is the program correct? If not, point out the assertion that fails.

```
fun lock(l)
  { requires(l=1
  );
  ensures(l=0);
  l := 0;
  return l;
}

fun unlock(l)
  { requires(l=0
  );
  ensures(l=1);
  l := 1;
  return l;
}
```

Global variables

Quiz:

- So far, we assumed our function call doesn't have side-effects
- But suppose our function f has access to some global variable g
- Does our method, as presented so far, still work?

Global variables

- To deal with global variables, we will make use of x := **havoc**()
- Extend method contracts:

```
fun f(x₁, …, xₙ)
  { requires(Pre);
  ensures(Post);
  modifies(v₁, v₂, …);
  s;
  return e;
}
```

- Need to check that indeed only v_1, v_2, … are modified

Global variables

- Given such a contract, we can translate a function call as follows:

$$x := f(e_1, \ldots, e_n)$$

assert$(Pre[e_1/x_1, \ldots, e_n/x_n])$;**havoc**$(v_1, v_2, \ldots)$; $x := tmp$; **assume**$(Post[tmp/ret, e_1/x_1, \ldots, e_n/x_n])$;

Quiz:

- What happens if we leave out the havoc statement?

Adding Pointers

- Next, let's add pointers to Nano

Statement: $s \ni \text{Stmt} ::= \; y := *x \text{ (load)} \mid \; *x := e \text{ (store)} \mid \; \ldots$

Quiz:

- How would we have to modify our state in order to add pointers to our semantics?

- Does the old Hoare rule for assignments still work?

Old Rule: Counterexample

Example: x := y; *y := 3; *x := 2; z := *y; assert(z = 3)

Quiz:

- Should the post-condition hold?
- What's the weakest pre-condition?
- What's the problem with our old proof rule?

Verification with Pointers

- As the previous example shows, the old rule for assignments doesn't work!
- Problem: Due to aliasing, an assignment *x := e

  can affect values of expressions beyond *x

- Treat the heap as a gigantic array μ that maps addresses to values
- That means, we need the theory of arrays & new rules for store and load

Rules for Loads and Stores

- Loads

$$\frac{}{\vdash \{Q[\mu[y]/x]\}\ x := *y\ \{Q\}}$$

- Stores

$$\frac{}{\vdash \{Q[\mu\langle x \triangleleft e\rangle/\mu]\}\ *x := e\ \{Q\}}$$

Revisiting our example: New Rules

<u>Example:</u> x := y; *y := 3; *x := 2; z := *y; assert(z = 3)

Quiz:

- What's the weakest pre-condition with our new rules?
- What if we change our assertion to assert(z=2)?

$$\overline{\vdash \{Q[\mu[y]/x]\}\ x := *y\ \{Q\}}$$

$$\overline{\vdash \{Q[\mu\langle x \triangleleft e\rangle/\mu]\}\ *x := e\ \{Q\}}$$

Verification with Pointers

Quiz:

- How do our array rules reason about aliasing?
- Why is this computationally expensive?

Verification with Pointers

- Optimization: use pointer analysis to partition μ into several smaller arrays that can't alias
- What about data-structures like linked-lists & trees?
- There's another logic for that: separation logic! A primer on this below:
- http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/Marktoberdorf11LectureNotes.pdf
- Unfortunately, is undecidable, so automation is hard.
- However, successfully applied by Facebook/Meta: see https://fbinfer.com/
- More reading on VCs with pointers: https://github.com/barghouthi/cs704/blob/master/notes/cs704-lec-04-19-2010.pdf

What's next

- Next lectures:
  - Finding inductive loop invariants!
  - First: Horn clauses (good way to represent conditions on loop invariants)
  - Algorithms to solve Horn clauses = find invariants automatically