Solving Horn Clauses

Klaus v. Gleissenthall

VRIJE UNIVERSITEIT AMSTERDAM

Where are we?

- We want to compute loop invariants
- Last lecture, we looked at <u>Horn clauses</u>
- Horn clauses represent constraints on unknown relations called **queries**
- We used weakest preconditions to translate programs into Horn clauses
- In particular, this translation uses queries to represent <u>loop invariants</u>

Where are we?

- We also looked at a solving algorithm based on function post
- post computes the strongest post-condition of a formula w.r.t. a set of variables
- Unfortunately, the simple algorithm may diverge for clauses whose solutions represent infinite sets of states
- Next, we'll look at how to fix this using <u>abstraction</u>

Where are we?

- What happens, if we apply our algorithm to the following clauses?

$$(1) \quad x = 1 \land n \geq 1 \rightarrow \boldsymbol{q}(x, n)$$

$$(2) \quad (\boldsymbol{q}(y, n) \land x = y + 1 \land y < n) \rightarrow \boldsymbol{q}(x, n)$$

$(1) \quad x = 1 \land n \geq 1 \rightarrow \mathbf{q}(x, n)$

$(2) \quad (\mathbf{q}(y, n) \land x = y + 1 \land y < n) \rightarrow \mathbf{q}(x, n)$

Where are we?

- What happens, if we apply our algorithm to the following clauses?

(1) $x = 1 \wedge n \geq 1 \rightarrow \boldsymbol{q}(x, n)$

(2) $(\boldsymbol{q}(y, n) \wedge x = y + 1 \wedge y < n) \rightarrow \boldsymbol{q}(x, n)$

- Our algorithm won't terminate!
- Problem, we're keeping too much information about $\boldsymbol{q}$ *(all possible states!)*
- Say, we want to prove that $\boldsymbol{q}(x, n) \wedge x \geq n \rightarrow x = n \wedge x \geq 0$
- Then all we need to know is that $x \geq 0$ *and* $x \leq n$ always hold
- Our missing ingredient is <u>abstraction</u>

Predicate Abstraction

- We assume we're given a set of <u>atomic predicates</u> **P** in our background theory

<u>Example:</u> For example, we could set $\mathbf{P} \triangleq \{x \geq 0, x \geq 1, y \geq -1\}$

- The predicates define a <u>finite vocabulary</u> for expressing the solution of ***q***

- We define <u>abstraction function</u> $\alpha(\varphi, \mathbf{P})$, that given formula φ computes the <u>strongest conjunction</u> of atomic predicates in **P**, such that $\varphi \Rightarrow \alpha(\varphi, \mathbf{P})$

$$\alpha(\varphi, \mathbf{P}) \triangleq \bigwedge \{ p \in \mathbf{P} \mid \varphi \Rightarrow p \}$$

- $\alpha(\varphi, \mathbf{P})$ gives us the strongest post-condition in our <u>finite vocabulary</u>

Predicate Abstraction

Quiz:

- Let $\mathbf{P} \triangleq \{x \geq 0,\ x \geq 1,\ y \geq -1\}$ and $\varphi \triangleq x=0 \land z=1$
- What's $\alpha(\varphi, \mathbf{P})$?
- What if we let $\varphi \triangleq x=1 \land z=0$?
- What if $\varphi \triangleq \bot$?
- What if $\varphi \triangleq \top$?

$\alpha(\varphi, \mathbf{P}) \triangleq \bigwedge\{ p \in \mathbf{P} \mid \varphi \Rightarrow p \}$

Abstract Post

- Let's now use abstraction function $\alpha(\varphi, \mathbf{P})$ to define an operator **post#**

- Like **post**, **post#**$(\varphi, x_1, \ldots, x_n)$ computes the strongest post-condition of φ in terms of $x_1, \ldots, x_n$

- But now, **post#** abstracts the condition by limiting it to the finite vocabulary **P**

$$\mathbf{post}^{\#}(\varphi, x_1, \ldots, x_n) \triangleq \alpha(\,\mathbf{post}(\varphi, x_1, \ldots, x_n), \mathbf{P})$$

Computing a Solution with Abstraction

Quiz:

$\mathbf{post}^{\#}(\varphi, x_1, \ldots, x_n) \triangleq \alpha(\ \mathbf{post}(\varphi, x_1, \ldots, x_n), \mathbf{P})$

- Let **P** $\triangleq \{x \geq 0,\ x \geq 1,\ y \geq -1\}$
- What is $\mathbf{post}^{\#}(x=y+1 \wedge y \geq -1, x)$?
- What is $\mathbf{post}^{\#}(x=y+1 \wedge y=0, x)$?
- What is $\mathbf{post}^{\#}(x=y+1 \wedge y=1, x)$?

Computing a Solution with Abstraction

- Instead of post, we can now use **post#** in our algorithm
- Say, we have some fixed set of predicates **P**
- Like before, we start with a solution Σ that maps every ***query*** to ⊥
- Pick any clause whose head is a query, that is, a clause of the form

$$\mathbf{p}(y_1, y_2) \wedge \mathbf{q}(y_1, y_2, y_3) \wedge \ldots \wedge \varphi \rightarrow \mathbf{r}(x_1, x_2, x_3)$$

- and compute

$$\mathrm{p} \triangleq (\mathbf{post\#}(\Sigma(\mathbf{p}) \wedge \Sigma(\mathbf{q}) \wedge \ldots \wedge \varphi, x_1 x_2 x_3))$$

- Then, if $\not\models \mathrm{p} \rightarrow \Sigma(\mathbf{r})$, set $\Sigma(\mathbf{r}) := (\Sigma(\mathbf{r}) \vee \mathrm{p})$

Computing a Solution with Abstraction

<u>Example:</u>

(1) $x = 1 \land n \geq 1 \rightarrow$ **q**(x, n)

(2) $($**q**$(y, n) \land x = y + 1 \land y < n) \rightarrow$ **q**(x, n)

- Let's fix predicates **P** $\triangleq \{x \geq 1,\ x \geq 0,\ y \geq -1,\ x \leq n\}$
- We start off with solution $\Sigma \triangleq \{$**q** $\rightarrow \bot\}$
- Let's pick the first clause $x = 1 \land n \geq 1 \rightarrow$ **q**(x)

Quiz:

- What's **post#**$(x = 1 \land n \geq 1,\ x,\ n)$?
- Does **post#**$(x = 1 \land n \geq 1,\ x,\ n) \Rightarrow \bot$ holds?

Computing a Solution with Abstraction

Example:

$$(1) \quad x = 1 \wedge n \geq 1 \rightarrow \mathbf{q}(x, n)$$

$$(2) \quad (\mathbf{q}(y, n) \wedge x = y + 1 \wedge y < n) \rightarrow \mathbf{q}(x, n)$$

- Let's fix predicates $\mathbf{P} \triangleq \{x \geq 1, x \geq 0, y \geq -1, x \leq n\}$
- Our new solution is $\Sigma \triangleq \{\mathbf{q} \rightarrow (x \geq 0 \wedge x \geq 1 \wedge x \leq n)\}$
- Next, let's pick clause 2
- What is $\mathbf{post}^{\#}(y \geq 0 \wedge y \geq 1 \wedge y \leq n \wedge y < n \wedge x = y + 1, x, n)$?
- Does $\mathbf{post}^{\#}(y \geq 0 \wedge y \geq 1 \wedge y \leq n \wedge y < n \wedge x = y + 1, x, n) \Rightarrow x \geq 0 \wedge x \geq 1 \wedge x \leq n$ hold?

Computing a Solution with Abstraction

Example:

$$(1) \quad x=1 \wedge n \geq 1 \rightarrow q(x, n)$$

$$(2) \quad (q(y, n) \wedge x=y+1 \wedge y<n) \rightarrow q(x, n)$$

- Let's fix predicates $\mathbf{P} \triangleq \{x \geq 1, x \geq 0, y \geq -1, x \leq n\}$
- We're done! Our solution is $\Sigma \triangleq \{q \rightarrow (x \geq 0 \wedge x \geq 1 \wedge x \leq n)\}$
- We can now check if our solution satisfies clause $\mathbf{h} \triangleq q(x, n) \wedge x \geq n \rightarrow x=n \wedge x \geq 0$

Quiz:

- Does $\Sigma \vDash \mathbf{h}$ hold?
- Would $\Sigma \vDash \mathbf{h}$ still work for predicates $\mathbf{P} \triangleq \{x \geq 1, x \geq 0, x \leq n\}$?
- Would about for $\mathbf{P} \triangleq \{x \geq 1, x \leq n\}$?
- Would about for $\mathbf{P} \triangleq \{y \geq -1, x \leq n\}$?

Picking Predicates

- The million dollar question: where do the predicates come from?
- Can't be solved in general, but we can mine them from pre-, post-conditions, and program
- For our example, we want to prove $x = n \wedge x \geq 0$
- If we decompose $x = n$ into $x \leq n$ and $x \geq n$ we get the right predicates
- This heuristics might of course fail!
- In this case, we can supply missing predicates as annotations

Which clauses to schedule?

- Let's take another look at our algorithm, it says: pick <u>any clause</u>
- Pick <u>any clause</u> whose head is a query, that is, a clause of the form

$$\boldsymbol{p}(y_1, y_2) \wedge \boldsymbol{q}(y_1, y_2, y_3) \wedge \ldots \wedge \varphi \rightarrow \quad \boldsymbol{r}(x_1, x_2, x_3)$$

- and compute

$$p \triangleq (\mathbf{post}^{\#}(\Sigma(\boldsymbol{p}) \wedge \Sigma(\boldsymbol{q}) \wedge \ \ldots \ \wedge \varphi, x_1, x_2, x_3))$$

- Then, if $\not\models$ p $\rightarrow \quad \Sigma(\boldsymbol{r}),$ set $\qquad \Sigma(\boldsymbol{r}) := (\Sigma(\boldsymbol{r}) \vee p)$

Quiz:

- Which clauses do we start with?
- After updating query $\boldsymbol{r}$, which clauses do we pick next?

Example: Computing a Solution

<u>Example:</u>

- Let's look at the following clauses

(1) $x=0 \wedge y=0 \rightarrow q(x, y, n, f)$

(2) $q(x', y', n, f) \wedge f=1 \wedge x=x'+1 \wedge y=y'+1 \wedge x'<n \rightarrow q(x, y, n, f)$

(3) $q(x', y', n, f) \wedge f\neq 1 \wedge x=x'+1 \wedge y=y'-1 \wedge x'<n \rightarrow q(x, y, n, f)$

(4) $q(x, y, n, f) \wedge f = 1 \rightarrow y\geq 0$

Quiz:

- What are we computing?

Example: Computing a Solution

Example:

- The clauses correspond to the following program:

```
x=0; y=0;
while(x < n){
  x := x+1;
  if(f=1){ y:=y+1;}
  else {y := y-1;}
}
assert(f = 1 → y≥0)
```

Example: Computing a Solution

<u>Example:</u>

- Let's look at the following clauses

(1) $x=0 \wedge y=0 \rightarrow q(x, y, n, f)$

(2) $q(x', y', n, f) \wedge f=1 \wedge x=x'+1 \wedge y=y'+1 \wedge x'<n \rightarrow q(x, y, n, f)$

(3) $q(x', y', n, f) \wedge f\neq 1 \wedge x=x'+1 \wedge y=y'-1 \wedge x'<n \rightarrow q(x, y, n, f)$

(4) $q(x, y, n, f) \wedge f = 1 \rightarrow y\geq 0$

- Let's pick predicates $\mathbf{P} \triangleq \{y\geq 0, f\neq 1\}$
- Which solution does our algorithm compute?
- Does the solution prove clause (4)?

(1) $x=0 \wedge y=0 \rightarrow$ $q(x, y, n, f)$

(2) $q(x', y', n, f) \wedge f=1 \wedge x=x'+1 \wedge y=y'+1 \wedge x'<n \rightarrow$ $q(x, y, n, f)$

(3) $q(x', y', n, f) \wedge f \neq 1 \wedge x=x'+1 \wedge y=y'-1 \wedge x'<n \rightarrow$ $q(x, y, n, f)$

(4) $q(x, y, n, f) \wedge f = 1 \rightarrow$ $y \geq 0$

(1) $x=0 \wedge y=0 \rightarrow \ q(x, y, n, f)$

(2) $q(x', y', n, f) \wedge f=1 \wedge x=x'+1 \wedge y=y'+1 \wedge x'<n \rightarrow \ q(x, y, n, f)$

(3) $q(x', y', n, f) \wedge f \neq 1 \wedge x=x'+1 \wedge y=y'-1 \wedge x'<n \rightarrow \ q(x, y, n, f)$

(4) $q(x, y, n, f) \wedge f = 1 \rightarrow \ y \geq 0$

(1) $x=0 \wedge y=0 \rightarrow q(x, y, n, f)$

(2) $q(x', y', n, f) \wedge f=1 \wedge x=x'+1 \wedge y=y'+1 \wedge x'<n \rightarrow q(x, y, n, f)$

(3) $q(x', y', n, f) \wedge f \neq 1 \wedge x=x'+1 \wedge y=y'-1 \wedge x'<n \rightarrow q(x, y, n, f)$

(4) $q(x, y, n, f) \wedge f = 1 \rightarrow y \geq 0$

Correctness

- Why is it correct to use **post#** instead of **post**?
- Only intuition, no proof!

Quiz:

- Say, we're using our algorithm with **post**.
- Let Σ_n be the solution after n steps
- What does Σ_n represent?
- Idea: solution $\Sigma_\#$ computed by **post#** <u>over-approximates</u> Σ_n for any n

Correctness

- Say we have a clause b $\triangleq$ $\boldsymbol{q} \rightarrow \quad \neg\Psi$
- Idea: if $\Sigma_{\#}$ and Ψ do not intersect, we know Ψ is not reachable

Quiz:

- What can we conclude if $\Sigma_{\#}$ and Ψ do intersect?

Termination

Quiz:

- Will our algorithm always terminate?
- This is important, because we want our algorithm to be predictable!

Complete Lattice

- We'll only discuss a sketch of the argument!
- DNF Formulas over the predicates form a complete lattice
- Lattice: there is partial order $\sqsubseteq$ (has to be reflexive, transitive, anti-symmetric)
- We define a $\sqsubseteq$ b iff: $a \Rightarrow b$
- Example: **P** $\triangleq \{f \neq 1,\ y \geq 0\}$

$$\top$$

$$f \neq 1 \lor y \geq 0$$

$$f \neq 1 \qquad\qquad\qquad y \geq 0$$

$$f \neq 1 \land y \geq 0$$

$$\bot$$

Complete Lattice

- a ⊑ b: a is more precise than b.
- ⊤ : we know nothing
- Complete: for each subset S there is element b s.t., a ⊑ b, for all a ∈ S
- Useful fact: all finite lattices are complete

$$\top$$

$$f \neq 1 \ \vee y \geq 0$$

$$f \neq 1 \qquad\qquad\qquad\qquad y \geq 0$$

$$f \neq 1 \ \wedge \ y \geq 0$$

$$\bot$$

Kleene's Fixed Point Theorem

- <u>Sketch</u>: Our algorithm only <u>moves upwards</u> in the lattice
- Put differently, in each iteration our algorithm describes a function that's <u>monotonic</u> on the lattice, that is, only moves up and never down
- Since there's only finitely many elements in the lattice, the function terminates

$\top$

$f \neq 1 \lor y \geq 0$

$f \neq 1$ $\qquad\qquad$ $y \geq 0$

$f \neq 1 \land y \geq 0$

$\bot$

Kleene's Fixed Point Theorem

- Formally justified by Kleene's fixed-point theorem
- Also says that our algorithm computes the least fixed point (wrt. $\sqsubseteq$)
- And that the fixed-point is unique
- Doesn't just work for our algorithm, but whenever the restrictions are met
- For more details, see: https://cs.au.dk/~amoeller/spa/spa.pdf

$\top$

$f \neq 1 \lor y \geq 0$

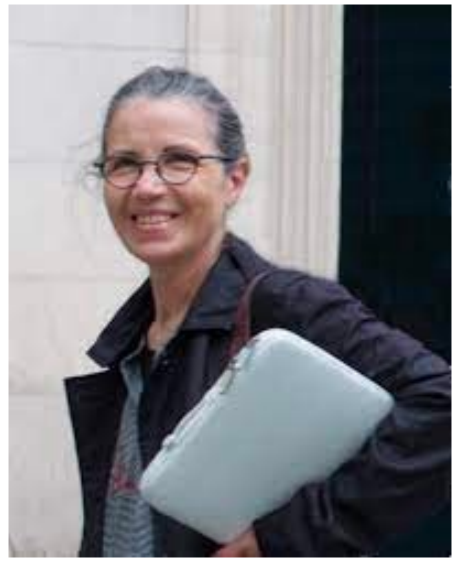
$f \neq 1$ $\qquad\qquad$ $y \geq 0$

$f \neq 1 \land y \geq 0$

$\bot$

Correctness: Abstract Interpretation

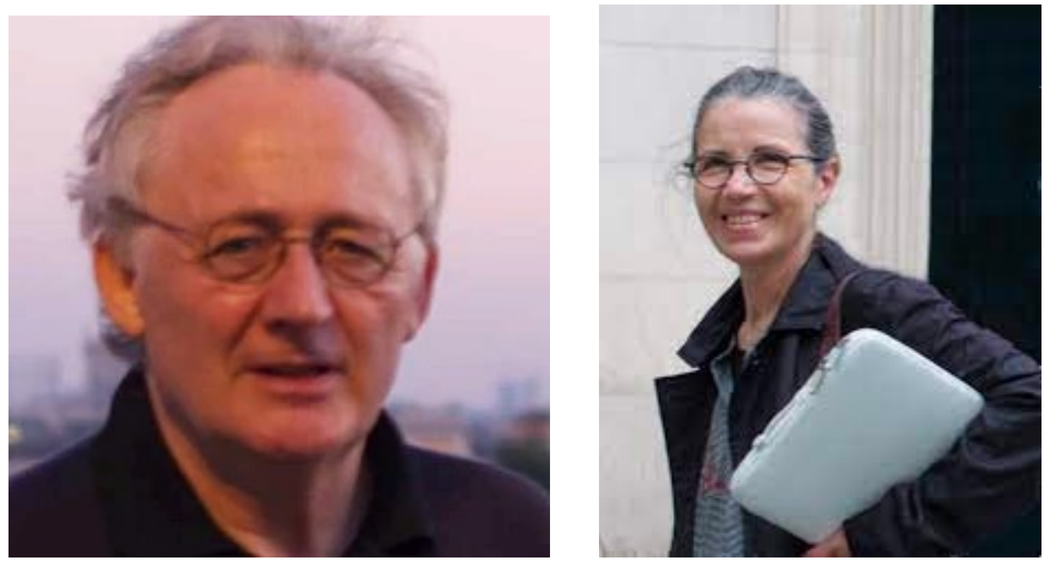
- We know that we over-approximate the real solution in each step, but is this enough?
- Not really!
- To fully justify the correctness of our algorithm we can use the theory of Abstract Interpretation
- Hugely influential theory
- Invented by Patrick and Radhia Cousot

Patrick and Radhia Cousot

Correctness: Abstract Interpretation

- Abstract interpretation works by defining two functions α and γ
- α called abstraction function and γ is called concretization function
- These functions link concrete solutions
  (as computed by the naive algorithm in FOL)
  and abstract solutions (expressed over predicates **P**)
  - α maps a concrete solution to an abstract solution
  - $\alpha(\varphi, \mathbf{P}) \triangleq \bigwedge \{ p \in \mathbf{P} \mid \varphi \Rightarrow p \}$
  - γ maps an abstract solution to a concrete solution
  - We can simply use the identity function for that!



Patrick and Radhia Cousot

Quiz:

- Why is that?

Correctness: Abstract Interpretation

- Abstract Interpretation then requires the two following equations to hold:

- For each abstract formula Ψ, it holds that $\alpha(\gamma(\Psi)) = \Psi$

- For each concrete formula φ, it holds that $\varphi \Rightarrow \gamma(\alpha(\varphi))$

- A pair of functions α, γ satisfying these equations is called Galois connection

- Together, these condition ensure that out algorithm computes an over-approximation of the concrete set of reachable states

- More on this:
- https://web.eecs.umich.edu/~weimerw/2006-615/reading/AbramskiAI.pdf

Correctness: Abstract Interpretation

- Abstract Interpretation then requires the two following equations to hold:

- For each abstract formula Ψ, it holds that $\alpha(\gamma(\Psi)) = \Psi$

- For each concrete formula φ, it holds that $\varphi \Rightarrow \gamma(\alpha(\varphi))$

Quiz:

- Do these condition hold ?

Abstraction Refinement

- There are also technique for <u>computing new predicates</u> from failed verification attempts
- Call a clause b $\triangleq$ ***q*** $\rightarrow$ Ψ a bound.
- Let's say our abstract solution $\Sigma^{\#}$ violates the bound $\Sigma^{\#}$, *i.,e.* $\Sigma^{\#} \not\models b$
- This might be because we're missing some information, i.e., some predicates
- We can rerun the computation with concrete post-condition to compute a solution Σ
- We can compute a formula φ s.t., $\Sigma(\mathbf{q}) \Rightarrow \varphi$ and $\varphi \Rightarrow \Psi$
- φ is called a <u>Craig interpolant</u> and captures the <u>reason</u> why Ψ holds on the real solution
- From φ we can extract missing predicates

Abstraction Refinement

- Does this really work?
- Sometimes! And it can be quite impressive
- But, there’s no guarantee that this algorithms will terminate
- This is no surprise! It’s trying to solve an undecidable problem.
- Unfortunately, this makes the algorithm <u>unpredictable</u>
- Will we find a solution in the next step, or will we loop forever?
- As we’ve seen, predicate abstraction <u>without</u> refinement is always guaranteed to terminate!

More

- We verifying imperative languages, yet, we're writing everything in Haskell
- A natural question is: can we also verify functional languages?
- We don't have to worry about state, but there's other complications:
  - Higher-order functions, closures, lazy evaluation etc.
- This can be done using refinement types:
  - https://arxiv.org/pdf/2010.07763.pdf
  - http://ucsd-progsys.github.io/lh-workshop/
- Refinement types generalize function contracts

What’s next?

- In the second assignment, you’ll get to implement the Horn clause solving approach
- You can decide if you think it helps verifying programs that need invariants
- For the lecture, this concludes the section on verification: now security
- We’ll first look at a type-system to stop sensitive information from leaking
- Last, we’ll look at speculative execution attacks and how to prevent them
- I’m also going to talk about my current research and how it uses all of this