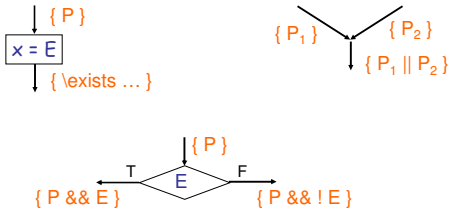
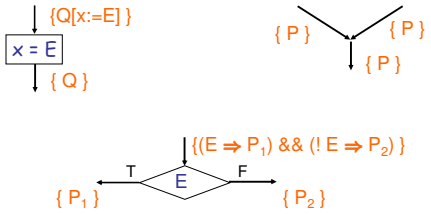
## Review: forward




## Review: backward

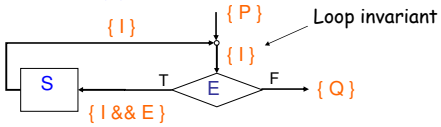



## ESC Verification algorithm

- Given function body annotated with pre-condition P and post-condition Q:
  - Compute wp of Q with respect to functon body
  - Ask a theorem prover to show that P implies the wp
- We saw several examples last time
- But we still haven't seen how to handle:
  - loops, functions calls, and pointers

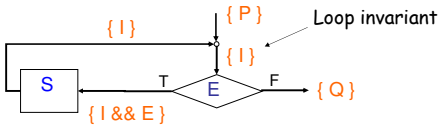
## Reasoning About Programs with Loops

- Loops can be handled using conditionals and joins
- Consider the while(E) S statement



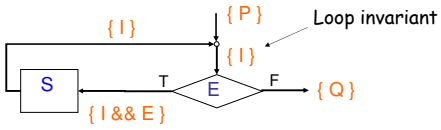

if (1) P ⇒ I (loop invariant holds initially)
and (2) I && ! E ⇒ Q (loop establishes the postcondition)
and (3) { I && E } S { I } (loop invariant is preserved)

## Loops in the backward direction




- Given Q, want to find weakest invariant I that will establish (2) and (3), then pick P to be I
- Finding weakest I is:
  - Undecidable in theory
  - Hard in practice

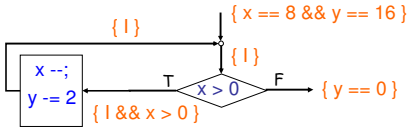
## Loops in the forward direction




- Given P, want to find strongest invariant I that will establish (1) and (3), then pick Q to be I && E
- Again, finding I is hard

## Loop Example

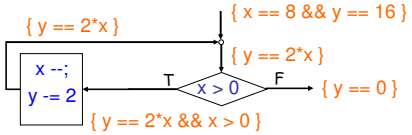
- Let's verify
  { x ==8 && y ==16 } while(x > 0) { x --; y -= 2; } { y==0 }
  – Is this true ?




  – We must find an appropriate invariant I
    - Try one that holds initially x == 8 && y == 16
    - Try one that holds at the end y == 0

## Loop Example (II)

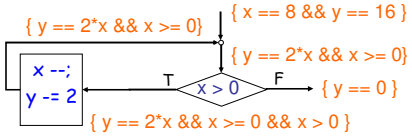
- Guess the invariant y == 2*x




- Must check
  – Initial: x == 8 && y == 16 ⇒ y == 2*x
  – Preservation: y == 2*x && x > 0 ⇒ y – 2 = 2*(x – 1)
  – Final: y == 2*x && x <= 0 ⇒ y == 0

## Loop Example (III)

- Guess the invariant y == 2*x && x >= 0




- Must check
  – Initial: x == 8 && y == 16 ⇒ y == 2*x && x >= 0
  – Preservation:
    y == 2*x && x >= 0 && x > 0 ⇒ y – 2 = 2*(x – 1) && x – 1 >= 0
  – Final: y == 2*x && x >= 0 && x <= 0 ⇒ y == 0

## Functions

- Consider a binary search function bsearch

```
int bsearch(int a[], int p) {
  { sorted(a) }                                   Precondition
  …
  { r == -1 || (r >= 0 && r < a.length && a[r] == p) }
  return res;                                     Postcondition
}
```

- The precondition and postconditon are the function specification
  – Also called a contract

## Function Calls

- Consider a call to function F(int in)
  – With return variable out
  – With precondition Pre, postcondition Post
- Rule for function call:

if P ⇒ Pre[in:=E] }

and Post[out := y, in := E] ⇒ Q

## Function Call Example

- Consider the call
  { sorted(array) }
  y = bsearch(array, 5)
  if( y != -1) {
  { array[ y ] == 5 }
- Show Post[r := y, a := array, p := 5]
  ⇒ array[y] == 5
  – Need to know y != -1 !
- Show sorted[array] ⇒ Pre[a := array]

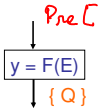
## Function Calls: backward

- Consider a call to function F(int in)
  - With return variable out
  - With precondition Pre, postcondition Post

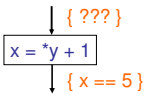
## Function Calls: backward

- Consider a call to function F(int in)
  - With return variable out
  - With precondition Pre, postcondition Post

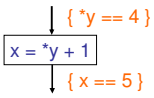
Pre[in:=E] ∧ Post[in:=E, out:=y] ⇒ Q




## Pointers and aliasing

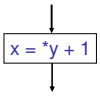



## Pointers and aliasing

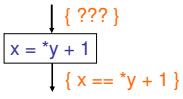



Regular rule worked in this case!

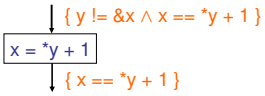
## Example where regular rule doesn't work



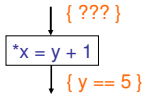

## Example where regular rule doesn't work



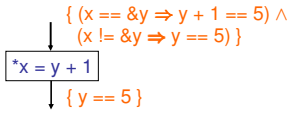
## Example where regular rule doesn't work




## Pointer stores

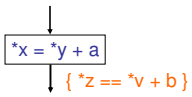



## Pointer stores




## One solution

- Perform case analysis based on all the possible alias relationships between the LHS of the assignment and part of the postcondition
- Can use a static pointer analysis to prune some cases out
- However, exponentially many cases in the pointer analysis, which leads to large formulas.
- eg, how many cases here:




## Another solution

- Up until now the program state has been implicit. Let's make the program state explicit...
- A predicate is a function from program states to booleans.
- So for wp(S, Q), we have:
  - Q(σ) returns true if Q holds in σ
  - wp(S, Q)(σ) returns true if wp(S, Q) holds in σ

## New formulation of wp

- Suppose step(S, σ) returns the program state resuling from executing S starting in program state σ.
- Then we can express wp as follows:

  wp(S, Q)(σ) =

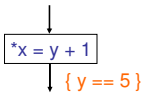
## New formulation of wp

- Suppose step(S, σ) returns the program state resuling from executing S starting in program state σ.
- Then we can express wp as follows:

  wp(S, Q)(σ) = Q(step(S, σ))

## Example in Simplify syntax

From previous slide: wp(S, Q)(σ) = Q(step(S, σ))



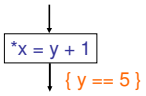

Q is: (EQ (select σ y) 5)

step(S, σ) is: (store σ (select σ x) (+ (select σ y) 1))

wp(S, Q) is:

## Example in Simplify syntax

From previous slide: wp(S, Q)(σ) = Q(step(S, σ))




Q is: (EQ (select s y) 5)

step(S, σ) is: (store s (select s x) (+ (select s y) 1))

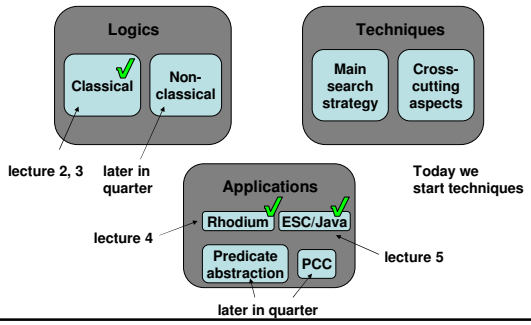
wp(S, Q) is:
(EQ (select (store s (select s x) (+ (select s y) 1)) y) 5)

## ESC/Java summary

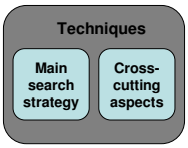
- Very general verification framework
  - Based on pre- and post-conditions
- Generate VC from code
  - Instead of modelling the semantics of the code inside the theorem prover
- Loops and procedures require user annotations
  - But can try to infer these

## Search techniques

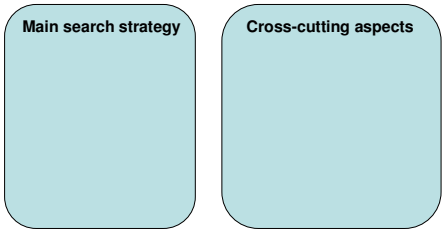
## The map



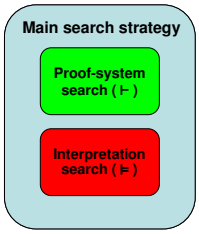
## Techniques in more detail




## Techniques in more detail



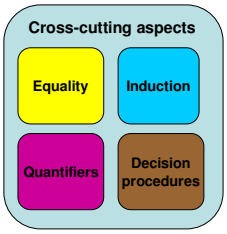

## Techniques in more detail



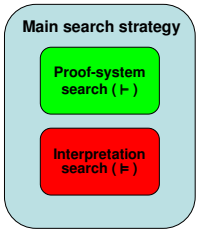

- Theorem proving is all about searching
- Categorization based on the search domain:
  - interpretation domain
  - proof-system domain

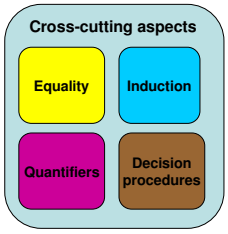
## Techniques in more detail

- Equality...
  - common predicate symbol
- Quantifiers...
  - need good heuristics
- Induction...
  - for proving properties of recursive structures
- Decision procedures...
  - useful for decidable subsets of the logic




## Techniques in more detail




## Searching

- At the core of theorem proving is a search problem
- In this course, we will categorize the core search algorithms based on what they search over
  - proof domain: search in the proof space, to find a proof
  - semantic domain: search in the "interpretation" domain, to make sure that there is no way of making the formula false
- Before we dive in, let's go back to some basic logic

## Logics

- Suppose we have some logic
  - for example, propositional logic

$$\phi ::= true \mid false \mid x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \Rightarrow \phi$$

  - or first-order logic

$$t ::= x \mid F(t, \ldots, t)$$
$$\phi ::= true \mid false \mid P(t, \ldots, t) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \Rightarrow \phi \mid \forall x.\phi \mid \exists x.\phi$$

## The two statements

$\Gamma \models \phi$ — set of formulas; one formula — "entails, or models"

In all worlds where the formulas in $\Gamma$ hold, $\phi$ holds

Semantic

$\Gamma \vdash \phi$ — "is provable from"

$\phi$ is provable from assumptions $\Gamma$

Syntactic

## Interpretations

- Intuitively, an interpretation $I$ represents the "world" in which you evaluate a formula
- Provides the necessary information to evaluate formulas
- The structure of $I$ depends on the logic
- Interpretations are also sometimes called models

## Interpretations in PROP

- Given a formula A $\wedge$ B , what do we need to evaluate it?
- We need to know the truth values of A and B
- In general, we need to know the truth values of all propositional variables in the formula
- Note that the logical connectives are built in, we don't have to say what $\wedge$ means

## Interpretations in FOL

- Given a formula: $\forall$ x. P(f(x)) $\Rightarrow$ P(g(x)), what do we need to know to evaluate it?
- We need to know how the function symbol f and predicate symbol P operate
- In general, need to know how all function symbols and predicate symbols operate
- Here again, logical connectives are built-in, so we don't have to say how $\Rightarrow$ operates.

## More formally, for PROP

- An interpretation $I$ for propositional logic is a map (function) from variables to booleans
  - So, for a variable A, $I$(A) is the truth value of A

## More formally, for FOL

- An interpretation for first-order logic is a quadruple (D, Var, Fun, Pred)
- D is a set of objects in the world
- Var is a map from variables to elements of D
  - So Var(x) is the object that variable x represents

## More formally, for FOL

- Fun is a map from function symbols to math functions
  - Fun(f) is the math function that the name f represents
  - For example, in the interpretation of LEQ(Plus(4,5), 10), we could have
    - D is the set of integers
    - Fun(4) = 4 , Fun(5) = 5 , Fun(10) = 10 , Fun(Plus) = +
    - But, we could also have Fun(Plus) = -
  - If f is an n-ary function symbol, then Fun(f) has type $D^n \rightarrow D$

## More formally, for FOL

- Pred is a map from predicate symbols to math functions
  - Pred(P) is the math function that the name P represents
  - For example, in the interpretation of LEQ(Plus(4,5), 10)
    - we could have Pred(LEQ) = <=
  - If P is an n-ary predicate, then Pred(P) has type $D^n \rightarrow$ bool

## Putting interpretations to use

- We write $[\![ \phi ]\!]_I$ to denote what $\phi$ evaluates to under interpretation $I$
- In PROP
  - $[\![ A ]\!]_I = I(A)$
  - $[\![ \neg \phi ]\!]_I$ = true iff $[\![ \phi ]\!]_I$ is not true
  - $[\![ \phi_1 \wedge \phi_2 ]\!]_I$ = true iff $[\![ \phi_1 ]\!]_I$ and $[\![ \phi_2 ]\!]_I$ are true
  - $[\![ \phi_1 \vee \phi_2 ]\!]_I$ = true iff $[\![ \phi_1 ]\!]_I$ or $[\![ \phi_2 ]\!]_I$ is true
  - etc.

## In FOL

- $[\![ x ]\!]_I$ = Var(x), where $I$ = (D, Var, Fun, Pred)
- $[\![ f(t_1, \ldots, t_n) ]\!]_I$ = Fun(f)($[\![ t_1 ]\!]_I, \ldots, [\![ t_n ]\!]_I$), where $I$ = (D, Var, Fun, Pred)
- $[\![ P(t_1, \ldots, t_n) ]\!]_I$ = Pred(P)($[\![ t_1 ]\!]_I, \ldots, [\![ t_n ]\!]_I$), where $I$ = (D, Var, Fun, Pred)
- Rules for PROP logical connectives are the same

## Quantifiers

- $[\![ \forall x . \phi ]\!]_{(D, Var, Fun, Pred)}$ = true iff

  forall o $\in$ D
  $[\![ \phi ]\!]_{(D, Var[x := o], Fun, Pred)}$ = true

- $[\![ \exists x . \phi ]\!]_{(D, Var, Fun, Pred)}$ = true iff

  there is some o $\in$ D for which
  $[\![ \phi ]\!]_{(D, Var[x := o], Fun, Pred)}$ = true

## Semantic entailment

- We write $\Gamma \models \phi$, where $\Gamma = \{\phi_1, \ldots \phi_n\}$, if for all interpretations $I$:
  - (Forall $i$ from 1 to n $[\![ \phi_i ]\!]_I$ = true) implies $[\![ \phi ]\!]_I$ = true
- For example
  - $\{ A \Rightarrow B, B \Rightarrow C\} \models A \Rightarrow C$
  - $\{ \} \models (\forall x.\ (P(x) \wedge Q(x))) \Leftrightarrow (\forall x.\ P(x) \wedge \forall x.\ Q(x))$
- We write $\models \phi$ if $\{ \} \models \phi$
  - we say that $\phi$ is a theorem

## Search in the semantic domain

- To check that $\models \phi$, iterate over all interpretations $I$ and make sure that $[\![ \phi ]\!]_I$ = true
- For propositional logic, this amounts to building a truth table
  - expensive, but can do better, for example using DPLL
- For first-order logic, there are infinitely many interpretations
  - but, by cleverly enumerating over Herbrand's universe, we can get a semi-algorithm

## Provability

- $\Gamma \vdash \phi$
- This judgement says that $\phi$ is provable from $\Gamma$
- Inference rules tell us how we can derive this judgement
- These inference rules are completely syntactic

## Some inference rules

$$\frac{}{\Gamma, A \vdash A}\ \text{Assume}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge E_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge E_2$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow I \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash A \Rightarrow B}{\Gamma \vdash B}\ \Rightarrow E$$

## A sample derivation

$$\dfrac{\dfrac{\dfrac{}{B \wedge A \vdash B \wedge A}\ \text{Assume}}{B \wedge A \vdash A}\ \wedge E_2 \qquad \dfrac{\dfrac{}{B \wedge A \vdash B \wedge A}\ \text{Assume}}{B \wedge A \vdash B}\ \wedge E_1}{B \wedge A \vdash A \wedge B}\ \wedge I$$

## Link between $\models$ and $\vdash$

- Soundness: $\Gamma \vdash \phi$ implies $\Gamma \models \phi$
- Completeness: $\Gamma \models \phi$ implies $\Gamma \vdash \phi$

- Virtually all inference systems are sound
- Therefore, to establish $\Gamma \models \phi$, all one needs to do is find a derivation of $\Gamma \vdash \phi$
- Can do this by searching in the space of proofs
  - forward, backward or in both direction

## Next class

- DPLL
- Herbrand's universe
- Davis-Putnam paper
- Explicating proofs paper