ESC Verification algorithm

- Given function body annotated with precondition $P$ and post-condition $Q$:
  - Compute wp of $Q$ with respect to function 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) $P \Rightarrow I$ (loop invariant holds initially)
  and (2) $I \& \& E \Rightarrow Q$ (loop establishes the postcondition)
  and (3) $(I \& \& E) S (I)$ (loop invariant is preserved)
Loop Example

- Let’s verify
  \( \{ x = 8 \land y = 16 \} \text{ while } (x > 0) \{ x--; y -= 2; \} \{ y = 0 \} \)
  - Is this true?
    \( \{ x = 8 \land y = 16 \} \)
    \( T \}
    \( F \}
    \( \{ y = 0 \} \)
    \( x > 0 \)
    \( \{ x = 8 \land y = 16 \} \)

- We must find an appropriate invariant \( I \)
  - Try one that holds initially \( x = 8 \land y = 16 \)
  - Try one that holds at the end \( y = 0 \)

Loop Example (II)

- Guess the invariant \( y = 2^x \)
  \( \{ y = 2^x \} \)
  \( \{ x = 8 \land y = 16 \} \)

- Must check
  - Initial: \( x = 8 \land y = 16 \Rightarrow y = 2^x \)
  - Preservation: \( y = 2^x \land x > 0 \Rightarrow y = 2^x(x-1) \)
  - Final: \( y = 2^x \land x <= 0 \Rightarrow y = 0 \)

Functions

- Consider a binary search function \( bsearch \)
  \( \text{int } bsearch(\text{int } a[], \text{int } p) \{ \)
  \( \text{sorted}(a) \)

Function Call Example

- Consider the call
  \( \{ \text{sorted}(array) \} \)
  \( y = bsearch(array, 5) \)
  \( \text{if} \ y \neq -1 \{ \)
  \( \{ \text{array}[y] = 5 \} \)
  \( \text{Post}[r := y, a := array, p := 5] \)

Function Calls

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

Rule for function call:

\( y = F(E) \)

\( \{ P \land \text{Pre}[\text{in} := E] \} \)

\( \{ Q \land \text{Post}[\text{out} := y, \text{in} := E] \} \Rightarrow Q \)
Function Calls: backward

• Consider a call to function $F(int \ in)$
  – With return variable $out$
  – With precondition $Pre$, postcondition $Post$

```
\begin{align*}
y = F(E) \quad \{Q\}
\end{align*}
```

Pointers and aliasing

```
\begin{align*}
x = \ast y + 1 \quad \{x == 5\}
\end{align*}
```

Example where regular rule doesn’t work

```
\begin{align*}
x = \ast y + 1
\end{align*}
```

Pointers and aliasing

```
\begin{align*}
x = \ast y + 1 \quad \{\ast y == 4\} \quad \text{Regular rule worked in this case!}
\end{align*}
```

Example where regular rule doesn’t work

```
\begin{align*}
x = \ast y + 1 \quad \{x == \ast y + 1\}
\end{align*}
```
Example where regular rule doesn’t work

\[
x = y + 1 \\
\{ x = y + 1 \}
\]

Pointer stores

\[
\begin{align*}
x &= y + 1 \\
\{ x = y + 1 \}
\end{align*}
\]

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:

\[
\begin{align*}
\ast x &= y + a \\
\{ \ast z &= \ast y + b \}
\end{align*}
\]

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:
  - \( Q(\sigma) \) returns true if \( Q \) holds in \( \sigma \)
  - \( wp(S, Q)(\sigma) \) returns true if \( wp(S, Q) \) holds in \( \sigma \)

New formulation of \( wp \)

- Suppose \( step(S, \sigma) \) returns the program state resulting from executing \( S \) starting in program state \( \sigma \).

- Then we can express \( wp \) as follows:
  \[
  wp(S, Q)(\sigma) = \]
New formulation of wp

• Suppose \( \text{step}(S, \sigma) \) returns the program state resulting from executing \( S \) starting in program state \( \sigma \).

• Then we can express \( \text{wp} \) as follows:
  \[
  \text{wp}(S, Q)(\sigma) = Q(\text{step}(S, \sigma))
  \]

Example in Simplify syntax

From previous slide: \( \text{wp}(S, Q)(\sigma) = Q(\text{step}(S, \sigma)) \)

\[
\begin{align*}
\text{x} &= \text{y} + 1 \\
\{ \text{y} == 5 \}
\end{align*}
\]

\( Q \) is: \((\text{EQ (select s y) 5})\)

\( \text{step}(S, \sigma) \) is: \((\text{store s (select s x) (+ (select s y) 1)})\)

\( \text{wp}(S, Q) \) is:

\((\text{EQ (select (store s (select s x) (+ (select s y) 1)) y) 5})\)

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

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

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

**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
    \[ \phi ::= \text{true} \mid \text{false} \mid x \mid \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid \phi \Rightarrow \phi \]
  – or first-order logic
    \[ t ::= x \mid F(t, \ldots, t) \]
    \[ \phi ::= \text{true} \mid \text{false} \mid P(t, \ldots, t) \mid \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid \phi \Rightarrow \phi \mid \forall x. \phi \mid \exists x. \phi \]

The two statements
\[
\Gamma \models \phi \quad \Gamma \vdash \phi
\]
– set of formulas
– one formula
– “entails, or models”
– “is provable from”

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

Semantic
Syntactic

Interpretations
• Intuitively, an interpretation \( \mathcal{I} \) represents the “world” in which you evaluate a formula
• Provides the necessary information to evaluate formulas
• The structure of \( \mathcal{I} \) depends on the logic
• Interpretations are also sometimes called models

Interpretations in PROP
• Given a formula \( A \land B \), 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 \( \land \) 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 \( \mathcal{I} \) for propositional logic is a map (function) from variables to booleans
  – So, for a variable \( A \), \( \mathcal{I}(A) \) 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 \( \text{LEQ}(\text{Plus}(4,5), 10) \), we could have
    • D is the set of integers
    • \( \text{Fun}(4) = 4 \), \( \text{Fun}(5) = 5 \), \( \text{Fun}(10) = 10 \), \( \text{Fun}(<) = - \)
    – But, we could also have \( \text{Fun}(\text{Plus}) = - \)
  – 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 \( \text{LEQ}(\text{Plus}(4,5), 10) \)
    – we could have \( \text{Pred}(\text{LEQ}) = \leq \)
  – If P is an n-ary predicate, then Pred(P) has type \( D^n \rightarrow \text{bool} \)

Putting interpretations to use

• We write \( [\phi]_I \) to denote what \( \phi \) evaluates to under interpretation \( I \)
  • In PROP
    – \( [A]_I = I(A) \)
    – \( [\neg \phi]_I \) is true iff \( [\phi]_I \) is not true
    – \( [\phi_1 \land \phi_2]_I \) is true iff \( [\phi_1]_I \) and \( [\phi_2]_I \) are true
    – \( [\phi_1 \lor \phi_2]_I \) is true iff \( [\phi_1]_I \) or \( [\phi_2]_I \) is true
    – etc.

In FOL

• \( [x]_I = \text{Var}(x) \), where \( \Gamma = (D, \text{Var}, \text{Fun}, \text{Pred}) \)
• \( [f(t_1, \ldots, t_n)]_I = \text{Fun}(f)([t_1]_I, \ldots, [t_n]_I) \), where \( \Gamma = (D, \text{Var}, \text{Fun}, \text{Pred}) \)
• \( [P(t_1, \ldots, t_n)]_I = \text{Pred}(P)([t_1]_I, \ldots, [t_n]_I) \), where \( \Gamma = (D, \text{Var}, \text{Fun}, \text{Pred}) \)
• Rules for PROP logical connectives are the same

Quantifiers

• \( [\forall x. \phi]_I(D, \text{Var}, \text{Fun}, \text{Pred}) = \text{true iff} \)
  
    for all \( o \in D \)
    \( [\phi]_I(D, \text{Var}[x := o], \text{Fun}, \text{Pred}) = \text{true} \)

• \( [\exists x. \phi]_I(D, \text{Var}, \text{Fun}, \text{Pred}) = \text{true iff} \)
  
    there is some \( o \in D \) for which
    \( [\phi]_I(D, \text{Var}[x := o], \text{Fun}, \text{Pred}) = \text{true} \)
Semantic entailment

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

Search in the semantic domain

- To check that $\vDash \phi$, iterate over all interpretations $I$ and make sure that $[ \phi ]_I = \text{true}$
- 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

- $\Gamma \vdash \phi$
- Assume $\Gamma, \alpha \vdash \beta$
  - $\Gamma, \alpha \vdash \beta \\
  - $\Gamma, \alpha \vdash \phi \\

A sample derivation

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

- Virtually all inference systems are sound
- Therefore, to establish $\Gamma \vDash \phi$, 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