Wei Hu Memorial Lecture

- I will give a completely optional bonus survey lecture: “A Recent History of PL in Context”
  - It will discuss what has been hot in various PL subareas in the last 20 years
  - This may help you get ideas for your class project or locate things that will help your real research
  - Put a tally mark on the sheet if you’d like to attend that day - I’ll pick a most popular day
- Likely Topics:

Today’s Cunning Plan

- Why Bother?
- Mathematical Induction
- Well-Founded Induction
- Structural Induction
  - “Induction On The Structure Of The Derivation”

Why Bother?

- I am loathe to teach you anything that I think is a waste of your time.
- Thus I must convince you that inductive opsem proof techniques are useful.
  - Recall class goals: understand PL research techniques and apply them to your research
  - This should also highlight where you might use such techniques in your own research.

Never Underestimate

“Any counter-example posed by the Reviewers against this proof would be a useless gesture, no matter what technical data they have obtained. Structural Induction is now the ultimate proof technique in the universe. I suggest we use it.” --- Admiral Motti, A New Hope

Classic Example (Schema)

- “A well-typed program cannot go wrong.”
  - Robin Milner
- When you design a new type system, you must show that it is safe (= that the type system is sound with respect to the operational semantics).
  - Type preservation: “if you have a well-typed program and apply an opsem rule, the result is well-typed.”
  - Progress: “a well-typed program will never get stuck in a state with no applicable opsem rules”
- Done for real languages: SML/NJ, SPARK ADA, Java
  - Plus basically every toy PL research language ever.
### Classic Examples

- **CCured Project (Berkeley)**
  - A program that is instrumented with CCured run-time checks ("adheres to the CCured type system") will not segfault ("the x86 opsem rules will never get stuck").
- **Vault Language (Microsoft Research)**
  - A well-typed Vault program does not leak any tracked resources and invokes tracked APIs correctly (e.g., handles IRQL correctly in asynchronous Windows device drivers, cf. Capability Calculus)
- **RC - Reference-Counted Regions For C (Intel Research)**
  - A well-typed RC program gains the speed and convenience of region-based memory management but need never worry about freeing a region too early (run-time checks).
- **Typed Assembly Language (Cornell)**
  - Reasonable C programs (e.g., device drivers) can be translated to TALx86. Well-typed TALx86 programs are type- and memory-safe.
- **Secure Information Flow (Many, e.g., Volpano et al. '96)**
  - Lattice model of secure flow analysis is phrased as a type system, so type soundness = noninterference.

### Recent Examples

- "The proof proceeds by rule induction over the target term producing translation rules."
  - Chakravarty et al. '05
- "Type preservation can be proved by standard induction on the derivation of the evaluation relation."
  - Hosoya et al. '05
- "Proof: By induction on the derivation of N ⊨ W."
  - Sumi and Pierce '05
- Method: chose four POPL 2005 papers at random, the three above mentioned structural induction.

### Induction

- **Most important technique for studying the formal semantics of prog languages**
  - If you want to perform or understand PL research, you must grok this!
  - Mathematical Induction (simple)
  - Well-Founded Induction (general)
  - Structural Induction (widely used in PL)

### Mathematical Induction

- **Goal**: prove \( \forall n \in \mathbb{N}. P(n) \)
  - **Base Case**: prove \( P(0) \)
  - **Inductive Step**:
    - Prove \( \forall n>0. p(n) \Rightarrow p(n+1) \)
    - "Pick arbitrary \( n \), assume \( p(n) \), prove \( p(n+1) \)"

### Why Does It Work?

- There are no infinite descending chains of natural numbers
- For any \( n \), \( P(n) \) can be obtained by starting from the base case and applying \( n \) instances of the inductive step

### Well-Founded Induction

- A relation \( \prec \subseteq A \times A \) is well-founded if there are no infinite descending chains in \( A \)
  - Example: \( \prec = \{(x, x+1) \mid x \in \mathbb{N}\} \)
  - the predecessor relation
  - Example: \( \prec = \{(x, y) \mid x, y \in \mathbb{N} \text{ and } x < y \} \)
- **Well-founded induction**:
  - To prove \( \forall x \in A. P(x) \) it is enough to prove \( \forall x \in A. [\forall y < x \Rightarrow P(y)] \Rightarrow P(x) \)
  - If \( \prec \) is \( \ll \), then we obtain mathematical induction as a special case
Structural Induction

- Recall $e ::= n \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid x$
- Define $\prec \subseteq \text{Aexp} \times \text{Aexp}$ such that
  
  - $e_1 \prec e_1 + e_2 \quad e_2 \prec e_1 + e_2$
  - $e_1 \prec e_1 \cdot e_2 \quad e_2 \prec e_1 \cdot e_2$

- no other elements of $\text{Aexp} \times \text{Aexp}$ are related by $\prec$

- **To prove** $\forall e \in \text{Aexp}. P(e)$
  
  1. $\vdash \forall n \in \mathbb{Z}. P(n)$
  2. $\vdash \forall x \in L. P(x)$
  3. $\vdash \forall e_1, e_2 \in \text{Aexp}. P(e_1) \land P(e_2) \Rightarrow P(e_1 + e_2)$
  4. $\vdash \forall e_1, e_2 \in \text{Aexp}. P(e_1) \land P(e_2) \Rightarrow P(e_1 \cdot e_2)$

Notes on Structural Induction

- Called **structural induction** because the proof is guided by the **structure** of the expression
- One proof case per form of expression
  
  - Atomic expressions (with no subexpressions) are all base cases
  
  - Composite expressions are the inductive case
- This is the most useful form of induction in PL study

Example of Induction on Structure of Expressions

- Let
  
  - $L(e)$ be the # of literals and variable occurrences in $e$
  - $O(e)$ be the # of operators in $e$

- Prove that $\forall e \in \text{Aexp}. L(e) = O(e) + 1$

- **Proof:** by induction on the structure of $e$
  
  - Case $e = n$. $L(e) = 1$ and $O(e) = 0$
  - Case $e = x$. $L(e) = 1$ and $O(e) = 0$
  - Case $e = e_1 + e_2$.
    
    - $L(e) = L(e_1) + L(e_2)$ and $O(e) = O(e_1) + O(e_2) + 1$
    - By induction hypothesis $L(e_1) = O(e_1) + 1$ and $L(e_2) = O(e_2) + 1$
    
    - Thus $L(e) = O(e_1) + O(e_2) + 2 = O(e) + 1$
  - Case $e = e_1 \cdot e_2$. Same as the case for $+$

Other Proofs by Structural Induction on Expressions

- Most proofs for Aexp sublanguage of IMP
- Small-step and natural semantics obtain equivalent results:
  
  - $\forall e \in \text{Exp}. \forall n \in \mathbb{N}. e \rightarrow^* n \iff e \uparrow n$

  - Structural induction on expressions works here because all of the semantics are syntax directed

Stating The Obvious (With a Sense of Discovery)

- You are given a concrete state $\sigma$.
- You have $\vdash <x + 1, \sigma> \Downarrow 5$
- You also have $\vdash <x + 1, \sigma> \Downarrow 88$
- Is this possible?

Why That Is Not Possible

- Prove that IMP is **deterministic**

- No immediate way to use mathematical induction
  
  - For commands we cannot use induction on the structure of the command
    
    - While $\text{else}$’s evaluation does not depend only on the evaluation of its strict subexpressions

    
    - $<b, \sigma> \Downarrow \text{true}$
    
    - $<c, \sigma \uparrow \sigma' \quad <\text{while } b \text{ do } c, \sigma > \Downarrow \sigma''$

    - While $b$ do $c$, $\sigma \uparrow \sigma''$
Recall Opsem

- **Operational semantics** assigns meanings to programs by listing rules of inference that allow you to prove judgments by making derivations.
- A derivation is a tree-structured object made up of valid instances of inference rules.

### Induction on the Structure of Derivations

- Key idea: The hypothesis does not just assume a \( c \in \text{Comm} \) but the existence of a derivation of \( \langle c, \sigma \rangle \Downarrow \sigma' \)
- Derivation trees are also defined inductively, just like expression trees
- A derivation is built of subderivations:
  - Adapt the structural induction principle to work on the structure of derivations

#### Induction on Derivations

- To prove that for all derivations \( D \) of a judgment, property \( P \) holds
  1. For each derivation rule of the form
     \[
     H_1 \ldots H_n \quad \frac{C}{C'}
     \]
  2. Assume that \( P \) holds for derivations of \( H_i \) (\( i = 1, \ldots, n \))
  3. Prove the the property holds for the derivation obtained from the derivations of \( H_i \) using the given rule

#### Induction on Derivations (2)

- Prove that evaluation of commands is deterministic:
  \[
  \langle c, \sigma \rangle \Downarrow \sigma' \Rightarrow \forall \sigma'' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma'' \Rightarrow \sigma' = \sigma''
  \]
- Pick arbitrary \( c, \sigma, \sigma' \) and \( D :: \langle c, \sigma \rangle \Downarrow \sigma' \)
- To prove: \( \forall \sigma'' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma'' \Rightarrow \sigma' = \sigma'' \)
- Proof: by induction on the structure of the derivation \( D \)
  - Case: last rule used in \( D \) was the one for skip
    \[
    D :: \frac{\text{skip \Downarrow \sigma'}}{\langle \text{skip}, \sigma \rangle \Downarrow \sigma'}
    \]
    - This means that \( c = \text{skip} \), and \( \sigma' = \sigma \)
    - By inversion \( \langle c, \sigma \rangle \Downarrow \sigma'' \) uses the rule for skip
    - Thus \( \sigma'' = \sigma \)
    - This is a base case in the induction

#### Induction on Derivations (3)

- Case: the last rule used in \( D \) was the one for sequencing
  \[
  D :: \begin{array}{c}
  D_1 :: \langle c_1, \sigma \rangle \Downarrow \sigma_1 \\
  D_2 :: \langle c_2, \sigma' \rangle \Downarrow \sigma'
  \end{array}
  \]
  - This means that \( c = \text{while} \), \( \sigma' = \sigma'' \)
  - By induction hypothesis on \( D_1 \) (with \( D_1' \)); \( \sigma_1 = \sigma'' \)
  - \( \text{while} D_1' :: \langle c_1, \sigma_1 \rangle \Downarrow \sigma'' \)
  - By induction hypothesis on \( D_2 \) (with \( D_1'' \)); \( \sigma'' = \sigma' \)
  - This is a simple inductive case

### New Notation

- Write \( D :: \text{Judgment} \) to mean “\( D \) is the derivation that proves \( \text{Judgment} \)”
- Example:
  \[
  D :: \langle x+1, \sigma \rangle \Downarrow 2
  \]
Induction on Derivations (4)

- Case: the last rule used in D was `while true`
  - Pick arbitrary $\sigma''$ such that $D :: \langle while b do c, \sigma \rangle \Downarrow \sigma''$
  - By inversion and determinism of boolean expressions, $D$ also uses the rule for `while true`
  - And has subderivations $D' :: \langle c, \sigma \rangle \Downarrow \sigma''$, and
  - $D'' :: \langle W, \sigma', \Downarrow \sigma'' \rangle$
  - By induction hypothesis on $D_1$ (with $D''): \sigma = \sigma''$
  - By induction hypothesis on $D_2$ (with $D''): \sigma' = \sigma''$

What Do You, The Viewers At Home, Think?

- Let’s do `if true` together!
- Case: the last rule in D was `if true`
  - Try to do this on a piece of paper. In a few minutes I’ll have some lucky winners come on down.

Induction on Derivations (5)

- Case: the last rule in D was `if true`
  - Pick arbitrary $\sigma''$ such that $D :: \langle if b do c1 else c2, \sigma \rangle \Downarrow \sigma''$
    - By inversion and determinism, $D$ also uses `if true`
    - And has subderivations $D'' :: \langle c, \sigma \rangle \Downarrow \sigma''$, and
    - $D'' :: \langle \sigma', \Downarrow \sigma'' \rangle$
  - By induction hypothesis on $D_1$ (with $D''): \sigma_1 = \sigma''$
  - Now $D_3 :: \langle while b do c, \sigma_1 \rangle \Downarrow \sigma'$
  - By induction hypothesis on $D_3$ (with $D''): \sigma'' = \sigma'$

Induction on Derivations Summary

- If you must prove $\forall x \in A. \; P(x) \Rightarrow Q(x)$
  - with A inductively defined and P(x) rule-defined
  - we pick arbitrary $x \in A$ and $D :: P(x)$
  - we could do induction on both facts
    - $x \in A$ leads to induction on the structure of $x$
    - $D :: P(x)$ leads to induction on the structure of $D$
    - Generally, the induction on the structure of the derivation is more powerful and a safer bet
  - Sometimes there are many choices for induction
    - choosing the right one is a trial-and-error process
    - a bit of practice can help a lot

Equivalence

- Two expressions (commands) are equivalent if they yield the same result from all states
  - $e_1 \approx e_2$ iff $\forall \sigma \in \Sigma. \forall n \in \mathbb{N}$. $<e_1, \sigma> \Downarrow n$ iff $<e_2, \sigma> \Downarrow n$
  - and for commands
    - $c_1 \approx c_2$ iff $\forall \sigma, \sigma' \in \Sigma$. $<c_1, \sigma> \Downarrow \sigma'$ iff $<c_2, \sigma> \Downarrow \sigma'$

Notes on Equivalence

- Equivalence is like logical validity
  - It must hold in all states (all valuations)
  - $2 = 1 + 1$ is like "$2 = 1 + 1$ is valid"
  - $2 = 1 + x$ might or might not hold.
    - So, 2 is not equivalent to $1 + x$
- Equivalence (for IMP) is undecidable
  - If it were decidable we could solve the halting problem for IMP. How?
- Equivalence justifies code transformations
  - compiler optimizations
  - code instrumentation
  - abstract modeling
- Semantics is the basis for proving equivalence
Equivalence Examples

• skip; c ≈ c
• while b do c ≈ if b then c; while b do c else skip
• If e₁ ≈ e₂ then x := e₁ ≈ x := e₂
• while true do skip ≈ while true do x := x + 1
• If c is while x = y do if x ≥ y then x := x · y else y := y · x then (x := 221; y := 527; c) ≈ (x := 17; y := 17)

Potential Equivalence

• (x := e₁; x := e₂) ≈ x := e₂
• Is this a valid equivalence?

Not An Equivalence

• (x := e₁; x := e₂) ∼ x := e₂
• Iie. Chigau yo. Dame desu!
• Not a valid equivalence for all e₁, e₂.
• Consider:
  - (x := x+1; x := x+2) ∼ x := x+2
• But for n₁, n₂ it’s fine:
  - (x := n₁; x := n₂) ≈ x := n₂

Proving An Equivalence

• Prove that "skip; c ≈ c" for all c
• Assume that D :: <skip; c, σ> ⇓ σ’
• By inversion (twice) we have that
  \[ D :: \frac{<\text{skip}, \sigma> \uparrow \sigma'}{D_1 :: <c, \sigma> \uparrow \sigma'} \]
• Thus, we have D₁ :: <c, σ> ⇓ σ’
• The other direction is similar

Proving An Inequivalence

• Prove that x := y ∼ x := z when y ≠ z
• It suffices to exhibit a σ in which the two commands yield different results
• Let σ(y) = 0 and σ(z) = 1
• Then
  \[ <x := y, \sigma> \uparrow \sigma[x := 0] \]
  \[ <x := z, \sigma> \uparrow \sigma[x := 1] \]

Summary of Operational Semantics

• Precise specification of dynamic semantics
  - order of evaluation (or that it doesn’t matter)
  - error conditions (sometimes implicitly, by rule applicability; “no applicable rule” = “get stuck”)
• Simple and abstract (vs. implementations)
  - no low-level details such as stack and memory management, data layout, etc.
• Often not compositional (see while)
• Basis for many proofs about a language
  - Especially when combined with type systems!
• Basis for much reasoning about programs
• Point of reference for other semantics
Homework

- Homework 1 Due Today
- Homework 2 Due Thursday
  - No more homework overlaps.
- Read Winskel Chapter 5
  - Pay careful attention.
- Read Winskel Chapter 8
  - Summarize.