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:

Proof Techniques
for Operational Semantics

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 motivation should also highlight where you might use such techniques in your own research.

Homework

- Use wrw6y or mst3k (etc.) not weimer
- Tuesday ends at midnight local time
- Wednesday Office Hours → Thursday
- Do not waste too much time on HW!
- Let’s do small-step opsem for “++x” together

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).
• A Syntactic Approach to Type Soundness. Andrew K. Wright, Matthias Felleisen, 1992.
  - 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
  - PL/I, 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 IRQs 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. (emphasis mine)

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 ∀n ∈ N. P(n)

  - Base Case: prove P(0)

  - Inductive Step:
    - Prove ∀ n>0, P(n) ⇒ 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 \( p \subseteq A \times A \) is well-founded if there are no infinite descending chains in \( A \).
  - Example: \( < \subseteq \mathbb{N} \times \mathbb{N} \) where \( x < y \) if \( x \in \mathbb{N} \) and \( x < y \).
  - aka the predecessor relation
  - Example: \( < \subseteq \mathbb{N} \times \mathbb{N} \) where \( x < y \) if \( x, y \in \mathbb{N} \) and \( x < y \).

Well-founded induction:

- To prove \( \forall x \in A. \; P(x) \) it is enough to prove \( \forall x \in A. \left( \forall y \in A. \; P(y) \right) \Rightarrow P(x) \).
- If \( p \) is \( < \), 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 \( \subseteq \subseteq \mathbb{N} \times \mathbb{N} \) such that:
  - \( e_1 p e_2 \) if \( e_1 + e_2 \)
  - \( e_1 p e_2 \) if \( e_1 \cdot e_2 \)
  - no other elements of \( \mathbb{N} \times \mathbb{N} \) are related by \( p \).

To prove \( \forall e \in \mathbb{N}. \; P(e) \):

1. \( \vdash \forall n \in \mathbb{N}. \; P(n) \)
2. \( \vdash \forall x \in \mathbb{N}. \; P(x) \)
3. \( \vdash \forall e_1, e_2 \in \mathbb{N}. \; P(e_1) \land P(e_2) \Rightarrow P(e_1 + e_2) \)
4. \( \vdash \forall e_1, e_2 \in \mathbb{N}. \; P(e_1) \land P(e_2) \Rightarrow P(e_1 \cdot e_2) \)

Example of Induction on Structure of Expressions

- Let \( \vdash P(e) \), where \( L(e) \) be the # of literals and variable occurrences in \( e \)
- \( O(e) \) be the # of operators in \( e \).

To prove \( \forall e \in \mathbb{N}. \; L(e) = O(e) + 1 \):

1. \( \vdash \forall n \in \mathbb{N}. \; P(n) \)
2. \( \vdash \forall x \in \mathbb{N}. \; P(x) \)
3. \( \vdash \forall e_1, e_2 \in \mathbb{N}. \; P(e_1) \land P(e_2) \Rightarrow P(e_1 + e_2) \)
4. \( \vdash \forall e_1, e_2 \in \mathbb{N}. \; P(e_1) \land P(e_2) \Rightarrow P(e_1 \cdot e_2) \)

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.

Other Proofs by Structural Induction on Expressions

- Most proofs for \( \mathbb{N} \) language of IMP.
- Small-step and natural semantics obtain equivalent results:
  \( \forall e \in \mathbb{N}. \; e \rightarrow^* n \iff e \in \mathbb{N} \).

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

\[ \forall e \in Aexp. \forall \sigma \in \Sigma. \forall n, n' \in N. <e, \sigma> \Downarrow n \land <e, \sigma> \Downarrow n' \Rightarrow n = n' \]

\[ \forall b \in Bexp. \forall \sigma \in \Sigma. \forall t, t' \in B. <b, \sigma> \Downarrow t \land <b, \sigma> \Downarrow t' \Rightarrow t = t' \]

• No immediate way to use mathematical induction

• For commands we cannot use induction on the structure of the command
  - while b's evaluation does not depend only on the evaluation of its strict subexpressions

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.

We Need Something New

• Some more powerful form of induction ...

• With all the bells and whistles!

Induction on the Structure of Derivations

• Key idea: The hypothesis does not just assume a c \in Comm but the existence of a derivation of <c, \sigma> \Downarrow \sigma'

• Derivation trees are also defined inductively, just like expression trees

• A derivation is built of subderivations:

\[ \text{while } x \leq 5 \text{ do } x := x + 1, \sigma \Downarrow \sigma' \]

New Notation

• Write \( D :: \text{Judgment} \) to mean “D is the derivation that proves Judgment”

Induction on Derivations

• To prove that for all derivations D of a judgment, property P holds

1. For each derivation rule of the form

\[ H_1 \ldots H_n \Rightarrow C \]

2. Assume 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: \(<c, \sigma> \uparrow \sigma^* \Rightarrow \forall \sigma^* \in \Sigma: <c, \sigma> \uparrow \sigma \Rightarrow \sigma^* = \sigma''\)
- Pick arbitrary \(c, \sigma, \sigma'\) and \(D :: <c, \sigma> \uparrow \sigma^*\)
- To prove: \(\forall \sigma^* \in \Sigma: <c, \sigma> \uparrow \sigma \Rightarrow \sigma^* = \sigma''\)
- Case: last rule used in \(D\) was the one for skip
  - Pick arbitrary \(c\)
  - Prove that evaluation of commands is deterministic:
    - To prove: 
      - \(D :: <c, \sigma> \uparrow \sigma^*\)
      - By induction hypothesis on \(D\)
      - This means that \(c = \text{skip}\), and \(\sigma^* = \sigma''\)
      - This is a base case in the induction.

**Induction on Derivations (3)**

- Case: the last rule used in \(D\) was the one for sequencing
  - Let’s do \(\text{if true}\) together!
  - Case: the last rule in \(D\) was \(\text{if true}\)
    - \(D ::<c_1, \sigma> \uparrow \sigma_1\), \(D_2 ::<c_2, \sigma> \uparrow \sigma_2\)
    - By induction hypothesis on \(D_1\) (with \(D_1'\)): \(\sigma_1 = \sigma_1''\)
    - Now \(D_1' ::<c_1, \sigma_1> \uparrow \sigma_1''\)
    - By induction hypothesis on \(D_2\) (with \(D_2'\)): \(\sigma_2'' = \sigma_2''\)
    - This is a simple inductive case

**Induction on Derivations (4)**

- Case: the last rule used in \(D\) was \(\text{while true}\)
  - \(D :: <\text{while b do c, } \sigma> \uparrow \sigma''\)
  - Pick arbitrary \(\sigma''\) such that \(D'' :: <\text{while b do c, } \sigma> \uparrow \sigma''\)
    - by inversion and determinism of boolean expressions, \(D''\) also uses the rule for \(\text{while true}\)
    - and has subderivations \(D''_1 :: <c, \sigma > \uparrow \sigma_1''\) and \(D''_2 :: <\text{while true, } \sigma_1''> \uparrow \sigma''\)
  - By induction hypothesis on \(D_2\) (with \(D_2''\)): \(\sigma_1 = \sigma_1''\)
    - Now \(D_2'' :: <\text{while b do c, } \sigma_1> \uparrow \sigma_1''\)
  - By induction hypothesis on \(D_1\) (with \(D_1''\)): \(\sigma'' = \sigma''\)

**Induction on Derivations (5)**

- Case: the last rule used in \(D\) was \(\text{if true}\)
  - \(D :: <\text{if b do c1 else c2, } \sigma> \uparrow \sigma''\)
  - Pick arbitrary \(\sigma''\) such that \(D'' :: <\text{if b do c1 else c2, } \sigma> \uparrow \sigma''\)
    - by inversion and determinism, \(D''\) also uses if true
    - And has subderivations \(D''_1 :: <b, \sigma > \uparrow \sigma_1''\) and \(D''_2 :: <\text{true, } \sigma_1''> \uparrow \sigma''\)
  - By induction hypothesis on \(D_2\) (with \(D_2''\)): \(\sigma'' = \sigma''\)

**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

**What Do You, The Viewers At Home, Think?**

- Let’s do \(\text{if true}\) together!
- Case: the last rule in \(D\) was \(\text{if true}\)
  - \(D :: <\text{if b do c, } \sigma> \uparrow \sigma_1\), \(D_2 :: <c_1, \sigma> \uparrow \sigma_2\)
  - Try to do this on a piece of paper. In a few minutes I’ll have some lucky winners come on down.
**Equivalence**

- Two expressions (commands) are equivalent if they yield the same result from all states
  \[ e_1 \approx e_2 \iff \forall \sigma \in \Sigma. \forall n \in \mathbb{N}. \langle e_1, \sigma \rangle \downarrow n \iff \langle e_2, \sigma \rangle \downarrow n \]

and for commands
  \[ c_1 \approx c_2 \iff \forall \sigma, \sigma' \in \Sigma. \langle c_1, \sigma \rangle \downarrow \sigma' \iff \langle c_2, \sigma \rangle \downarrow \sigma' \]

**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 \approx c`
- `while b do c ≈ if b then c; while b do c else skip`
- `if e_1 \approx e_2` then `x := e_1 \approx x := e_2`
- `while true do c \approx 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) \approx (x := 17; y := 17)`

**Potential Equivalence**

- `(x := e_1; x := e_2) \approx x := e_2`
- Is this a valid equivalence?

**Not An Equivalence**

- `(x := e_1; x := e_2) \not\approx x := e_2`
- `lie. Chigau yo. Dame desu!`
- Not a valid equivalence for all `e_1, e_2`.
- Consider:
  - `(x := x+1; x := x+2) \approx x := x+2`
  - But for `n_1, n_2` it's fine:
    - `(x := n_1; x := n_2) \approx x := n_2`

**Proving An Equivalence**

- Prove that "`skip; c \approx c`" for all `c`
- Assume that `D :: <skip; c, \sigma> \Downarrow \sigma'`
- By inversion (twice) we have that
  \[ D :: \langle \text{<skip, } \sigma \rangle \cup \sigma' \]
- Thus, we have `D_1 :: \langle \langle c, \sigma \rangle \Downarrow \sigma' \rangle`
- The other direction is similar
Proving An Inequivalence

• Prove that \( x := y \not\sim x := z \) when \( y \neq z \)

• It suffices to exhibit a \( \sigma \) in which the two commands yield different results

• Let \( \sigma(y) = 0 \) and \( \sigma(z) = 1 \)

• Then
  \[
  \langle x := y, \sigma \rangle \Downarrow \sigma[x := 0] \\
  \langle x := z, \sigma \rangle \Downarrow \sigma[x := 1]
  \]

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.