CS655 Lecture 4 | Proof Techniques for Operational Semantics | 31 January, 2006

Scribe: Jinlin Yang
Lecturer: Wes Weimer

Wes started today's lecture with an announcement of an optional lecture on history of PL based on Wei Hu's request. Wes planned to spend one hour introducing different programming language research projects with an focus on their advantages, disadvantages, and relationships.

Today's topic was *induction*, a proof technique that is widely used in modern PL research. After a long motivation section on the importance of induction, we covered three types of induction: mathematical induction, well-founded induction, and structural induction. Structural induction was today's focus.

1 Why Study Induction?

- Andrew K. Wright and Matthias Felleisen popularized and formalized the use of structural induction as a way to prove the soundness of a type system (by proving type preservation and progress theorems). A typical soundness proof shows that a well-typed program "cannot go wrong" in some sense. For example, Pascal programs cannot have segmentation faults.
- Many practical programming languages such as SPARK ADA and (Featherweight) Java have such soundness proofs. In addition, almost every toy PL research language (or language feature) has such a proof.
- We will use induction extensively in the class. So we need to understand structural induction to be able to understand the materials in the class.
- Many PL research projects use it. Examples chosen at random include: CCured, Vault, RC, Typed Assembly Language, and some projects involving Secure Information Flow.
- PL researchers still actively use it. For example, three of four randomly chosen papers in POPL 2005 mentioned structural induction. Induction is definitely indispensable for understanding the latest PL research.

In summary, induction is the most important proof technique in the study of formal language semantics.

2 Mathematical Induction

In *mathematical induction*, our proof target ranges over the natural numbers. Our goal is to prove the validity of our proof target for any natural number. Formally, we want to prove

$$\forall\, n \in \mathbb{N}.\ P(n)$$

The proof process consists of the two familiar steps. We first prove the base case $P(0)$. Then in the inductive step, we assume $P(n)$ for any arbitrarily picked n and prove $P(n+1)$. Mathematical induction works because there are no infinite descending chains of natural numbers.

Mathematical induction only proves properties of natural numbers. Computer programs, however, do not range over the natural number. Therefore, we cannot directly apply mathematical induction to prove properties of a program.

3 Well-Founded Induction

Well-founded induction generalizes mathematical induction to non-$\mathbb{N}$ domains.

Definition 1 *A relation $\prec \subseteq A \times A$ is well-founded if there are no infinite descending chains in A.*

Example:

$$\langle_1 \;=\; \{(x, x+1) \;\mid\; x \in \mathbb{N}\}$$

The idea of well-founded induction is similar to mathematical induction but uses the $\prec$ relation to determine the set of induction hypotheses at each inductive step.

Formally, in well-founded induction

$$\text{to prove } \forall\, x \in A.P(x), \text{ we need to prove } \forall\, x \in A.[\forall\, y \prec x \Rightarrow P(y)] \Rightarrow P(x)$$

Mathematical induction is a special case of the well-founded induction when $\prec$ is $\langle_1$. In summary, the well-founded induction is a general form of induction that is applicable to non-$\mathbb{N}$ domains.

4 Structural Induction

Structural induction is a special case of well-founded induction where the $\prec$ relation is defined on the structure of a program or a derivation.

4.1 Induction on a recursive definition

Induction on a recursive definition proves a property about a mathematical structure by demonstrating that the property holds for all possible forms of that structure.

4.1.1 Definition of the $\prec$ relation for IMP arithmetic expressions

The grammar of IMP arithmetic expressions is shown below

$$e ::= n \mid e_1 + e_2 \mid e_1 * e_2 \mid x$$

Therefore, we define the $a \prec b$ if a is a substructre of b. We defined the $\prec$ relation for IMP arithmetic expressions as follows:

$$\prec \;\subseteq\; \mathsf{Aexp} \;*\; \mathsf{Aexp} \;\; \text{such that}$$

$$e_1 \prec e_1 + e_2 \qquad e_1 \prec e_1 * e_2 \qquad e_2 \prec e_1 + e_2 \qquad e_2 \prec e_1 * e_2$$

4.1.2 The proof process

The proof process consists of proving that the property holds for the base cases (e is a base case if there is no e' such that $e' \prec e$) and also proving that the property holds for the inductive cases (e is an inductive case if $\exists e' \,.\, e' \prec e$).

4.1.3 Example

Let $L(e)$ be the number of literals and variable occurrences in e and $O(e)$ be the number of operators in e. Prove that $\forall e \in$ Aexp. $L(e) = O(e) + 1$

Proof: The proof is by induction on the structure of e.

Base cases:

- $e = n$. $L(e) = 1$ and $O(e) = 0$.
- $e = x$. $L(e) = 1$ and $O(e) = 0$.

Inductive cases:

- $e = e_1 + e_2$

  By definition we have $L(e) = L(e_1) + L(e_2)$ and $O(e) = O(e_1) + O(e_2) + 1$. By the induction hypothesis (on e_1 and e_2) we have $L(e_1) = O(e_1) + 1$ and $L(e_2) = O(e_2) + 1$. Thus $L(e) = O(e_1) + O(e_2) + 2 = O(e) + 1$.

- $e = e_1 * e_2$

  Same as the case for $+$.

4.2 Induction on the structure of a derivation

As a motivating example, we would like to prove that every IMP program is deterministic. Formally, this means

$$\begin{array}{lll} \forall e \in \mathsf{Aexp}.\ \forall \sigma \in \Sigma.\ \forall n, n' \in \mathbb{N}. & \langle e, \sigma\rangle \Downarrow n \ \wedge\ \langle e, \sigma\rangle \Downarrow n' & \Rightarrow \quad n = n' \\ \forall b \in \mathsf{Bexp}.\ \forall \sigma \in \Sigma.\ \forall t, t' \in \mathbb{B}. & \langle b, \sigma\rangle \Downarrow t \ \wedge\ \langle b, \sigma\rangle \Downarrow t' & \Rightarrow \quad t = t' \\ \forall c \in \mathsf{Comm}.\ \forall \sigma, \sigma', \sigma'' \in \Sigma. & \langle c, \sigma\rangle \Downarrow \sigma' \ \wedge\ \langle c, \sigma\rangle \Downarrow \sigma'' & \Rightarrow \quad \sigma' = \sigma'' \end{array}$$

We cannot use induction on the definition of Comm to prove determinism because the evaluation of the while command does not depend only on the evaluation of its strict subexpressions.

4.2.1 The proof process

Because induction on the structure of commands cannot prove the determinism property for IMP program, we changed the domain of induction. We adapted the structural induction principle to work on the structure of derivations. To prove a property P holds, we will prove that P holds for all possible derivations of a judgment.

Such a proof consists of the following steps:

- *Base cases*: for each atomic derivation rule of the form

  $$\frac{}{C}$$

  we have to show that property P holds.

- *Inductive cases*: for each derivation rule of the form

  $$\frac{H_1 \ldots H_n}{C}$$

  By the induction hypothesis, P holds for H_i, where $i = 1 \ldots n$. We have to prove that the property is preserved by the derivation using the given rule of inference.

We define $D :: \mathsf{Judgment}$ to mean "D is the derivation that proves Judgment". A key technique we will use extensively in the induction on derivations is *inversion*. Since the number of forms of rules of inference is finite, we can tell which inference rules might have been used last in the derivation. For example, given $D :: \langle x := 55, \sigma\rangle \Downarrow \sigma'$ we know (by inversion) that the assignment rule of inference must be the last rule used in D (because no other rules of inference involve an assignment command in their concluding judgment). Similarly, if $D :: \langle whilebdoc, \sigma\rangle \Downarrow \sigma'$ then (by inversion) the last rule used in D was either the while true rule or the while false rule.

4.2.2 Proof of determinism for IMP commands

Our goal is to prove the following property

$$\forall c \in \mathsf{Comm}.\ \forall \sigma, \sigma', \sigma'' \in \Sigma.\ \langle c, \sigma\rangle \Downarrow \sigma' \ \wedge\ \langle c, \sigma\rangle \Downarrow \sigma'' \Rightarrow \sigma' = \sigma''$$

We define $D\ ::\ \langle c, \sigma\rangle \Downarrow \sigma'$.

Proof: we prove the above property by induction on the structure of derivation D.

- *Base cases*: when the last rule used in D was the one for skip. I.e.,

$$D \;::\; \frac{}{\langle\, \mathsf{skip},\, \sigma\rangle \Downarrow \sigma}$$

  By inversion, the last rule used in $D' :: \langle c, \sigma\rangle \Downarrow \sigma''$ must also be skip. By the structure of the skip rule we know that $\sigma = \sigma''$, so the property holds.

- *Inductive cases*: we need to show that the property holds when the last rule used in D was each of the possible non-skip IMP commands. Here I will show a representative proof when the last rule used in D was the while true command

$$D \;::\; \frac{D_1 :: \langle b, \sigma\rangle \Downarrow \mathsf{true} \quad D_2 :: \langle c, \sigma\rangle \Downarrow \sigma_1 \quad D_3 :: \langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma_1\rangle \Downarrow \sigma'}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle \Downarrow \sigma'}$$

  Pick arbitrary σ'' such that $D'' :: \langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle \Downarrow \sigma''$

  By inversion and determinism of boolean expressions, we know D'' must also use the same rule for while true. So D'' must has subderivations ${D_2}'' :: \langle c, \sigma\rangle \Downarrow {\sigma_1}''$ and ${D_3}'' :: \langle \mathsf{while}\ b\ \mathsf{do}\ c, {\sigma_1}''\rangle \Downarrow \sigma''$

  By the induction hypothesis on D_2 with ${D_2}''$, we have $\sigma_1 = {\sigma_1}''$. Using this result and the induction hypothesis on D_3 with ${D_3}''$, we have $\sigma'' = \sigma'$.

4.3 Equivalence and Inequivalence

Two expressions (commands) are equivalent ($\approx$) if they yield same result from all states:

$$e_1 \approx e_2 \text{ iff } \forall \sigma \in \Sigma.\ \forall n \in \mathbb{N}.\ \langle e_1, \sigma\rangle \Downarrow n \text{ iff } \langle e_2, \sigma\rangle \Downarrow n$$

and for commands

$$c_1 \approx c_2 \text{ iff } \forall \sigma, \sigma' \in \Sigma.\ \langle c_1, \sigma\rangle \Downarrow \sigma' \text{ iff } \langle c_2, \sigma\rangle \Downarrow \sigma'$$

Equivalence is very useful for a variety of code transformation tasks. Unfortunately equivalence for IMP is undecidable ($c \approx c$ iff c halts).

To prove the equivalence of two expressions (commands), we could also use induction on their derivations. To prove the inequivalence of two expressions (commands), it is sufficient to demonstrate a counter- example.

4.4 Summary

Suppose you must prove $\forall x \in A.P(x) \Rightarrow Q(x)$. (For example, A might be Comm and $P(x)$ and $Q(x)$ might be operational semantics judgments). If A is inductively defined and the inference rules of $P(x)$ are inductively defined, we could either induction on the structure of $x \in A$ or induction on the structure of the derivation $D :: P(x)$. In general, the induction on the structure of the derivation is more powerful and a safer bet.

5 Summary of Operational Semantics

- Precise specification of dynamic semantics
- Simple and abstract
- Often not compositional. For example, the while command
- Basis for many proofs about a language
- Basis for much reasoning about programs
- Point of reference for other semantics